

# **Inference of Program Properties with Attribute Grammars, Revisited**

**Met attributtengrammatica's heen, met  
programma-eigenschappen terug**

(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof.dr. G.J. van der Zwaan, ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op maandag 9 januari 2012 des middags te 12.45 uur

door

**Adriaan Middelkoop**

geboren op 8 februari 1983 te Gorinchem

Promotor: Prof.dr. S. D. Swierstra  
Co-promotor: Dr. A. Dijkstra

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics) and was supported by Microsoft Research through its European PhD Scholarship Programme.

Author: Adriaan Middelkoop, 2011  
Printed by: Wöhrmann Print Service  
Cover photo: Fushimi Inari by Stéfan Le Dù, 2008

# Preface

With Attribute Grammars (AGs), Knuth [1968] presented a formalism that turned out to be excellent for reasoning about computations that can be expressed as a traversal over a tree. It is surprising how many programs can be conceptually described as an attribute grammar. On the other hand, it may be difficult to precisely formulate a program as such. In particular, several aspects of type inference algorithms are not straightforward to describe with an AG because these aspects require fixpoint iteration combined with dynamic tree construction in their implementation. In this thesis, we address this issue, and this thesis is thus a potpourri of type systems and AGs, with the greater focus being on the latter.

Recent attribute grammar systems compute attributes in a demand-driven evaluation order determined at runtime, which is relatively straightforward to implement when incorporating or embedding AGs in some other programming language [Sloane et al., 2010]. We, on the other hand, resurrect ordered attribute grammars, for which there exists a static order in which the attributes can be computed [Kastens, 1980]. This concept played an important role in the past to obtain efficient implementations when every byte of memory and every clock tick mattered.

For us, however, another consequence of the existence of a static order is important. In a recent paper, Kaminski and Van Wijk [2011] show the need to express in the AG description a distinction between trees that have not been attributed yet and trees that have been. Owing to the static order, we go a step further and distinguish and make assumptions about trees in various stages of their attribution. We show that this provides a means to describe, control and orchestrate chunks of attribute evaluation, which we exploit to fine-tune AG evaluation to capture inference algorithms.

With this thesis, we hope to show the power of AGs to implementors of type inference algorithms, and shed a new light on ordered AGs for implementors of AGs.

**Acknowledgements.** My promotor Doaitse Swierstra and copromotor Atze Dijkstra were a crucial source of inspiration. Despite his busy life, Doaitse takes the time to spread his brilliant ideas and interesting anecdotes. Doaitse is a great advocate of functional programming. We had many interesting discussions, although these usually started with an account of the low-level details of the tools that he fiddled with the day before. Or the latest functional programming exams. Nevertheless, Doaitse sets a great example, and it was an honor to work under his supervision.

Atze sets another example, especially when the exploration of a design space is concerned, and the decomposition into orthogonal aspects. It is amazing how Atze approaches problems with clear and concrete solutions, yet reasons at a high level of abstraction. Atze has a calm appearance, yet I also experienced some of his adventurous and active traits. In particular, I recall early-morning training runs during conferences, at times that others were deep asleep.

## Preface

The reading committee deserves a compliment for bravely reading through my thesis, as well as the anonymous reviewers before them that had to go through the papers from which this thesis is composed.

The Software Technology group at Utrecht is a pleasant group to be part of, and I thank all my colleagues and former colleagues for the enjoyable time. I especially recall the crazy group effort for the ICFP contest in 2007.

I thank my former office mates Eelco Dolstra, José Pedro Magalhães, and Stefan Holdermans. I hope that Stefan does not tear his eyes out if he spots errors in the Dutch summary at the back of this thesis. I also had pleasant discussions with my room mates Jeroen Bransen and Alexander Elyasov. I hope that you will enjoy attribute grammars as much as I do.

Halfway in my thesis project, I worked for half a year in Brazil. I want to heartily thank Lucília Camarão de Figueiredo for her effort in making my stay in Brazil pleasant.

The last year, I worked together with Wishnu Prasetya and Jurriaan Hage to statically instrument ActionScript bytecode. Jurriaan creates a warm atmosphere in the group, and is always around for a conversation or two.

The Universität Freiburg, and the ProgLang group in particular, generously providing me with a place to work each time I visited Freiburg. *Herzlichen Dank!* Also to Peter Thiemann for some heavy sport training sessions after work.

Thanks to Andres Löh for taking the initiative, we have with several colleagues a weekly distraction after work in the form of the Ars Magica games that we are still playing remotely. It is surprising what hidden personalities colleagues expose when exploring a fantasy setting.

The rather secretive IRC channel #klaploppers, formed by former occupants of the ST-Lab, provided welcoming distractions, both online and in real life. I want to thank Arthur van Dam and others for several nice mountainbike trips, although I'm not insane enough on a bike to be a real challenge. Together with Martin Bravenboer, we still have an Ironman Triathlon to finish. Eelco Dolstra, a living index of Wikipedia [*citation needed*], Rob Vermaas, Dick Eimers, my greetings to you. Last but not least, Armijn Hemel saved the IRC channel countless of times from my incorrect spelling and grammar.

Words fail to describe the wonderful role that Annette Bieniusa plays in my life. I tried once, but did not come close. Not even with 91 pages of emails. She may appear cute, small and innocent, but in the meantime she stole my heart. She is my greatest source of happiness and support. Although she cleverly managed to keep attribute grammars out of our discussions, she bravely read through my thesis and gave me valuable comments.

Tot slot wil ik mijn ouders en zusjes bedanken. Ik heb jullie tijdens het schrijven van dit werk niet veel aandacht kunnen geven, en ik hoop in de komende tijd daar wat verandering in aan te brengen, ondanks dat ik nu wat verder weg woon.

**Sponsors.** I am grateful to Microsoft Research in Cambridge for granting me a Ph. D. Scholarship, which made my employment at the Universiteit Utrecht possible. Also, the European project LerNet and the Universidade Federal de Minas Gerais partially financed me so that I could work for half a year in Brazil. Many ideas in this thesis arose during my daily walks to and from the university in Belo Horizonte.

# Contents

<b>Preface</b>	<b>1</b>
<b>1 Introduction</b>	<b>7</b>
1.1 Overview	7
1.2 Background on Type Systems	9
1.2.1 Specification of Programming Languages	10
1.2.2 The Lambda Calculus	10
1.2.3 Type Rules	11
1.2.4 Type Inference	14
1.2.5 Parametric Polymorphism	18
1.2.6 Damas-Hindley-Milner Inference	20
1.2.7 Polymorphic Lambda Calculus	22
1.2.8 Discussion	23
1.3 Background on Attribute Grammars	24
1.3.1 Syntax of Context-Free Grammars and Attribute Grammars	24
1.3.2 Dependency Graphs	30
1.3.3 Tree-Walking Automata	34
1.3.4 Demand-driven Attribute Evaluation	37
1.3.5 Statically Ordered Attribute Evaluation	39
1.3.6 Incremental Descriptions	40
1.3.7 Higher-Order Children and Attributes	42
1.3.8 Circular Reference Attributes	43
1.3.9 Correspondences between AGs, HOAGs, Monads, and Arrows	44
1.3.10 Specification of Typing Relations	45
1.3.11 Damas-Hindley-Milner Inference	46
1.3.12 Copy Rules and Collection Attributes	48
1.3.13 Advantages and Disadvantages	51
1.4 Background on Ruler	51
1.4.1 Ruler Features	52
1.4.2 Ruler Concepts	52
1.4.3 Damas-Hindley-Milner Inference	56
1.4.4 Discussion	56
1.5 Thesis Overview	57
1.5.1 Inference Algorithms as an Attribute Grammar	58
1.5.2 Attribute Grammar Extensions	58
1.5.3 Contextual Chapters	60

## Contents

1.6	The Context of this Thesis . . . . .	61
1.6.1	Challenges . . . . .	61
1.6.2	Additional Challenges . . . . .	62
1.6.3	Solutions . . . . .	63
1.7	Related Work . . . . .	64
1.7.1	Circular AGs and Exposure of Intermediate States . . . . .	64
1.7.2	Inference Rules . . . . .	64
1.7.3	Proof Assistants . . . . .	64
1.7.4	Ott . . . . .	65
1.7.5	TinkerType . . . . .	66
1.7.6	Overview of Recent Attribute Grammar Systems . . . . .	69
1.8	Conclusion . . . . .	69
<b>2</b>	<b>Outline of the RulerCore Concepts</b>	<b>71</b>
2.1	Attribute Grammars with Side Effects . . . . .	71
2.2	Attribute Grammars with Commuting Rules . . . . .	81
2.3	AGs with Tree Construction . . . . .	87
2.4	Case Study with GADTs . . . . .	95
2.5	Attribute Grammars with Stepwise Evaluation . . . . .	100
2.6	Attribute Grammars with Dependent Types . . . . .	107
2.7	Attribute Grammars on DAGs . . . . .	110
2.8	Conclusion . . . . .	110
<b>3</b>	<b>AGs with Side Effects</b>	<b>111</b>
3.1	Introduction . . . . .	111
3.2	Example . . . . .	113
3.2.1	Visitor Design Pattern . . . . .	113
3.2.2	Attribute Grammars . . . . .	115
3.2.3	RulerCore . . . . .	120
3.2.4	Desugared RulerCore . . . . .	124
3.3	Static Semantics of RulerBack . . . . .	127
3.4	Translation of RulerBack to JavaScript . . . . .	131
3.5	Translation of RulerCore to RulerBack . . . . .	133
3.5.1	Implicit Invocations . . . . .	133
3.5.2	Rule Ordering . . . . .	135
3.6	Discussion . . . . .	141
3.7	Related Work . . . . .	141
3.8	Conclusion . . . . .	142
<b>4</b>	<b>AGs with Commuting Rules</b>	<b>145</b>
4.1	Introduction . . . . .	145
4.2	Example with Barriers . . . . .	147
4.3	Core Representation of AGs with Barriers . . . . .	149
4.4	Static Dependency Graphs . . . . .	154

4.5	Visits Graphs . . . . .	156
4.6	Optimizations . . . . .	168
4.7	Execution Plans and Generated Code . . . . .	170
4.8	Generalization to Phases . . . . .	176
4.9	Commuting Rules . . . . .	182
4.10	Related Work . . . . .	186
4.11	Conclusion . . . . .	187
<b>5</b>	<b>Derivation Tree Construction</b>	<b>189</b>
5.1	Introduction . . . . .	189
5.2	Motivation . . . . .	190
5.2.1	Example: the Shadow-language . . . . .	191
5.2.2	Relation to Attribute Grammars . . . . .	192
5.2.3	Typing Expressions . . . . .	193
5.2.4	Unification . . . . .	199
5.2.5	Lookups in the Environment . . . . .	202
5.2.6	Translation to the Target Expression . . . . .	206
5.2.7	Discussion . . . . .	207
5.3	Semantics . . . . .	208
5.3.1	Syntax . . . . .	208
5.3.2	Example . . . . .	209
5.3.3	Translation . . . . .	211
5.4	Related Work . . . . .	212
5.5	Conclusion . . . . .	214
<b>6</b>	<b>AGs with Stepwise Evaluation</b>	<b>215</b>
6.1	Introduction . . . . .	215
6.2	Example of a Stepwise AG for a Predicate Language . . . . .	217
6.2.1	Syntax of the Predicate Language . . . . .	218
6.2.2	Deterministic Operational Semantics . . . . .	218
6.2.3	Declarative Operational Semantics . . . . .	219
6.2.4	Stepwise Operational Semantics . . . . .	220
6.2.5	Hybrid On-demand and Stepwise Evaluation . . . . .	223
6.3	SAG Translation . . . . .	223
6.4	Lazy Coroutines and the Stepwise Monad . . . . .	225
6.5	Imperative Implementation . . . . .	227
6.6	Remarks . . . . .	228
6.6.1	Extensions . . . . .	228
6.6.2	Benchmarks . . . . .	228
6.7	Related Work . . . . .	229
6.8	Conclusion . . . . .	231
6.A	Progress Reports and their Emission . . . . .	231
6.B	Translation Scheme . . . . .	232
6.C	Semantic Lookahead . . . . .	234

## Contents

6.D	Watchers . . . . .	235
<b>7</b>	<b>AGs with Dependent Types</b>	<b>237</b>
7.1	Introduction . . . . .	237
7.2	Preliminaries . . . . .	238
7.3	Dependently Typed Example . . . . .	241
7.3.1	Support Code Dealing With Environments . . . . .	241
7.3.2	Grammar of the Source and Target Language . . . . .	242
7.3.3	Dependent Attributes . . . . .	243
7.3.4	Semantics of Attributes . . . . .	244
7.4	AG Descriptions and their Core Representation . . . . .	245
7.5	Translation to Agda . . . . .	247
7.6	Partially Defined Attributes . . . . .	251
7.7	Related Work . . . . .	252
7.8	Conclusion . . . . .	253
7.A	Implementation of the Support Code . . . . .	254
7.A.1	Absurd Rules . . . . .	256
7.B	Dependent Nonterminal Attribution . . . . .	258
7.C	Ideas Transferrable to AG Systems for Haskell . . . . .	261
<b>8</b>	<b>Conclusion</b>	<b>263</b>
8.1	Addressed Challenges . . . . .	263
8.2	Solutions . . . . .	264
8.3	Remarks . . . . .	265
8.4	Implementations . . . . .	265
8.5	Future Work . . . . .	266
	<b>Bibliography</b>	<b>267</b>
	<b>Samenvatting</b>	<b>275</b>



# 1 Introduction

This thesis investigates the application of attribute grammars to the description of inference algorithms (as implemented in a compiler) that are specified by a collection of inference rules. Such collections are a means to specify the semantics of programming languages. Since there exists no universal inference algorithm for inference rules, it makes the description of inference algorithms a nontrivial exercise. Our goal is to use attribute grammars to make it easier to write such descriptions.

## 1.1 Overview

**Language specifications.** A formal programming language specification describes the language's notation and the notation's meaning, and forms a set of requirements for tools that process programs written in this language. A static semantics of a programming language is often expressed as a relation between properties of programs written in this language, where the relation is described by a set of inference rules. The structure of such a program is the Abstract Syntax Tree (AST) obtained by parsing the source code of the program, and plays usually an important role in the rules. For example, in case of code generation, a static semantics expresses a relation between the AST and machine code, whereas in case of type checking, the semantics typically relates the AST to a type for each program fragment and an environment with a type for each identifier.

**Implementations.** A compiler analyzes a program and computes properties of that program. This analysis is specified by a *static semantics*, which describes a relation between programs and properties. For example, the specification of a compiler that translates source code to machine code describes a relation between a program and machine instructions. The properties computed by a compiler can be seen as the evidence that there exists a proof that the properties are related to the program. Assuming that the relations are specified with inference rules, such a proof has the form of an attributed tree. Each node represents an application of an inference rule of the semantics, and attributes describe how the rules are instantiated.

A semantics is *syntax directed* if the shapes of the proofs are isomorphic to the AST. A semantics is *algorithmic* if additionally the constraints between attributes in the tree are expressible as computable functions. As a compiler's implementation is usually based on traversals of the AST, an algorithmic semantics is a convenient specification of a compiler. Since the implementation boils down to traversing the tree and applying the computable functions as mentioned in the specification in the right order, there exists a clear correspondence between the specification and the implementation. An example is a semantics for an assembly language where the relation between the AST and machine code is actually a bijection.

## 1 Introduction

However, most language's semantics are not entirely algorithmic. We come back to this point further below.

Attribute Grammars (AGs) [Knuth, 1968] are an attractive domain specific programming language for the implementation of an algorithmic semantics. Attributes represent inductive properties of the AST. A context-free grammar describes when an AST is correctly structured by relating productions to nodes in the AST. An AG describes when an AST is correctly decorated with attributes by imposing constraints on the attributes per node of the AST, formalized by rules per production. These rules are computable functions between attributes denoted in some host language. AGs abstract from the traversal over the AST and from the order of evaluation of the rules. Therefore, AGs are composable, which makes it possible to describe the implementation of a large language as various separate aspects. Moreover, an AG is compilable to an efficient algorithm that computes the attributes. Thus, an AG serves both as a specification and an implementation of an algorithmic semantics.

**Declarative specifications.** As argued above, to implement a static semantics, it is preferable that the semantics is algorithmic because of the close correspondence with an implementation. A declarative semantics is a semantics suitable for formal reasoning and documentation purposes, and is usually more abstract and concise than an algorithmic semantics.

For example, consider a static semantics that describes a relation between the AST and a sequence of machine instructions. When the relation is not functional, there may be many related sequences of machine instructions for a given AST. In particular, there may be a difference between a shortest sequence of instructions, or a sequence of instructions with the lowest expected execution time. The choice of which sequence is computed is left up to the implementation or specified separately.

Moreover, declarative features of a programming language can usually be more concisely described relationally instead of being based on a computable function. An example is operator overloading, where the choice of the implementation of an operator depends on the types of the operands. Effectively, the compiler takes care of some of the work of the programmer, and models the inference of the proof in some way.

**From ASTs to proofs.** A transformation from a non-algorithmic semantics to an algorithmic semantics is non-trivial. An implementation using traversals of the AST is based on an algorithmic semantics. Consequently, it is difficult to keep the implementation consistent with the declarative language specification.

In this thesis, we approach the implementation of a static semantics from another direction. We consider AGs based on the grammar that underlies the inference rules of the relation instead of the language's grammar. As a particular advantage, the AG is closely related to the declarative specification. However, the AG specifies when an attributed proof is valid, but does not specify how to obtain the proof. Since there exists no general procedure that maps such an AG to an algorithm that infers the proof, we need to augment the AG with additional information to obtain such an algorithm.

**Thesis.** In this thesis, we focus on the description of type inference algorithms with attribute grammars. Most type inference algorithms are a complex combination of a small set of inference techniques, such as type variables and unification to calculate with values that are not fully known yet, fixpoint iteration to approximate solutions, constraints to defer the inference of a subtree, and search tree construction to encode alternative solutions. Such techniques are based on gradually building a proof and exploring intermediate candidate solutions.

In our approach, we conservatively extend AGs to support these techniques and balance between assumptions about the evaluation algorithm, and the preservation of the declarative nature of the description. The central concept that underlies our extensions are higher-order, ordered attribute grammars. In such higher-order AGs, the domain of an attribute can be an attributed tree, which allows us to dynamically grow the tree. The state of a tree is described by a configuration, which specifies the decorations that have been computed. In an ordered AG, the configurations are linearly ordered. A visit, a unit of evaluation for a node, transitions the state of the node to a state described by the next configuration. This concept offers control over the simultaneous evaluation of attributes and exploration of the tree.

**Chapter organization.** This chapter presents background information and a short outline. We assume that the reader has a strong background in the programming language Haskell [O’Sullivan et al., 2008], is familiar with type systems [Pierce, 2002], and knows the basics of attribute grammars [Knuth, 1968]. This chapter gives (rather) informal definitions of relevant concepts and provides pointers to literature.

The actual contents of the thesis start with the next chapter, Chapter 2. Chapter 2 gives a detailed summary of our extensions and shows how these fit together. Each following chapter covers an extension in detail.

We address concepts of type systems and attribute grammars in combination with notation in Section 1.2 and Section 1.3. In particular, we recast the notion of ordered attribute grammars. Section 1.4 addresses previous work, Section 1.5 gives an overview of each of the extensions, and Section 1.6 sketches the overall goal. Finally, Section 1.7 addresses related work.

## 1.2 Background on Type Systems

In later chapters, we use type systems based on variants of the lambda calculus as example. In this section, we give a short summary on the lambda calculus, show the evaluation of lambda terms, and give a type system. Furthermore, the discussion of variants of the lambda calculus and their type systems serves as a vehicle to discuss design and implementation challenges of type systems, which we use in the motivation of this thesis in Section 1.5. Another purpose of this section is to introduce vocabulary and notation for subsequent chapters.

We assume that the reader is already familiar with the lambda calculus and type systems. Introductory books on type systems [Pierce, 2002, Harper, 2010] provide a more extensive and formal explanation.

## 1.2.1 Specification of Programming Languages

Programming languages are described by a grammar since a grammar specifies the set of programs that belong to the language. A semantics relates a program to some properties in a given domain. When these properties represent the correctness of the program, or represent a program in (another) programming language, we talk about a static or denotational semantics. When the properties describe the runtime behavior of the program (typically in the form of state transitions), we talk about a dynamic or operational semantics.

We assume that the specification of a programming language consists of a context-free grammar (Section 1.3.1) and static semantics. The specified programming language is called the *object language*, and a program an *object term*. In case of compilation and transformation, the object language is often referred to as the *source language* and object terms as *source terms*. The language that describes the specification and the language in which the compiler is implemented are called *meta languages*. Finally, in case of a translation to a different language, the language where we translate to is called the *target language* and the translated program a *target term*.

## 1.2.2 The Lambda Calculus

The lambda calculus is a language that is often used in programming language research, and in research on type systems in particular. Many concepts of programming languages have their roots in variants of the lambda calculus, or have been well-studied in such a context. We discuss the lambda calculus, because we use its concepts in object languages, source languages and target languages in the following chapters.

Figure 1.1 shows the abstract syntax  $e$  of expressions in an explicitly-typed variant of the simply-typed lambda calculus, which may contain types  $\tau$ . In passing, we also give syntax for environments and evaluation contexts. The structure of  $e$  is called the Abstract Syntax Tree when represented as a tree (Section 1.3.1).

In the lambda calculus, a function may be passed as a value  $v$  in the form of a lambda expression, and thus is *first class*. Such a function is anonymous, although the function can be given an explicit name by binding the function to an identifier. For example, the following expression denotes the application of a function that applies the identity function that it receives as an argument to the value 3:

$$(\lambda y : I \rightarrow I . y \ 3) (\lambda x : I . x) \quad \text{-- evaluates to 3 of type } I$$

To specify to what value an expression evaluates, we provide below an operational semantics. In this semantics, the simultaneous substitution of all free  $x$  in  $e_1$  by  $e_2$  is denoted by  $[x := e_2] e_1$ . The semantics<sup>1</sup> consists of the reduction relation  $e_1 \rightsquigarrow e_2$ , which is the smallest relation that satisfies the following inference rules<sup>2</sup> of Figure 1.2, which we explain below.

---

<sup>1</sup> Such a semantics is called a small-step operational semantics because the inference rules describe a transformation step from an intermediate term to another intermediate term, and the actual transformation is the exhaustive application of these transformation steps.

<sup>2</sup> In Section 1.2.3 we will actually consider a notation for inference rules and their interpretation.

$e ::= e_1 e_2$	-- application, with expression $e_1$ and $e_2$
$x$	-- variables, e.g. $x, y, z$
$v$	-- values (head-normal form)
$v ::= i$	-- integer constant, e.g. 3 and 42
$\lambda x : \tau. e$	-- abstraction, with param $x$ of type $\tau$ and body $e$
$\tau ::= I$	-- integer type
$\tau_1 \rightarrow \tau_2$	-- function type
$\Gamma ::= \emptyset$	-- empty environment
$\Gamma, x : \tau$	-- environment $\Gamma$ , with on top a mapping of $x$ to type $\tau$
$x, y, z$	-- variables
$f, g, h, a, e$	-- expressions

**Figure 1.1:** Syntax of the explicitly-typed lambda calculus variant.

$e_1 \rightsquigarrow e_2$	
$(\lambda x : \tau. e_1) e_2 \rightsquigarrow [x := e_2] e_1$	BETA
	$\frac{e_1 \rightsquigarrow e_2}{e_1 e \rightsquigarrow e_2 e}$ LEFT

**Figure 1.2:** Operational semantics as a reduction relation on expressions  $e$ .

A *beta-redex* is an expression of the form  $(\lambda x. e) a$  which can be reduced. The above rules describe normal-order reduction. Through the rule LEFT the beta-redex in the head position is identified. Indeed, if we consider the reduction of the above example, we end up first substituting  $f$  with the identity function, such that we obtain  $(\lambda x : I. x) 3$ , and then substitute 3 for  $x$ , such that we end up with the value 3. When no reductions are possible anymore, the expression is in *head normal form*.

### 1.2.3 Type Rules

The purpose of a static semantics is to exclude programs that incorrectly use their data. A type system classifies expressions as well-typed if it can associate a type with it. A type system is sound if it has the subject reduction property, which means that after each reduction step, the resulting term has the same type as the original expression. Type systems typically ensure the absence of certain programming errors, such as passing an integer where a function is expected.

As an example, we give a type system specification for the lambda calculus as defined above in Figure 1.4 and explain it below. We allow liberal syntactic sugar in our specifica-

## 1 Introduction

$r ::= \forall \bar{x} . d$	-- quantified rule over meta variables $\bar{x}$
$d ::= j_1 \dots j_n ; j$	-- rule with premisses $\bar{j}$ , conclusion $j$ , $n \geq 0$
$j ::= R^n \bar{m}$	-- judgment with $ \bar{m}  = n$
$R^n$	-- $n$ -place relation $R$ ( $n$ often omitted)
$m$	-- argument (meta expression)
$x, y$	-- meta variables for values in the object language

The notation  $\frac{j_1 \dots j_n}{j}$  (with optional rule label) is sugar for  $j_1 \dots j_n ; j$ . The explicit equality  $x_1 \approx x_2$  denotes a judgment  $Eq_\tau^2 x_1 x_2$ , with relation  $Eq_\tau$  representing structural equality of object terms of meta-type  $\tau$ . The quantification of meta variables is usually left implicit.

**Figure 1.3:** Notation for inference rules.

$\Gamma \vdash e : \tau$			
$\Gamma \vdash i : I$ CON	$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$ VAR	$\frac{\Gamma \vdash f : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash a : \tau_1}{\Gamma \vdash f a : \tau_2}$ APP	$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda(x : \tau_1).e : \tau_1 \rightarrow \tau_2}$ LAM

**Figure 1.4:** Type rules of the explicitly typed lambda calculus variant.

tions. Depending on the context, the notation  $\bar{x}$  represents a sequence or set  $x_1, \dots, x_n$ .

We define a typing relation  $\Gamma \vdash e : \tau$ , which is notation for a three-place relation over  $\Gamma$ ,  $e$ , and  $\tau$ . A judgment of this relation states that in environment  $\Gamma$ , the expression  $e$  has type  $\tau$ . We typically describe such relations with inference rules (type rules). Figure 1.3 gives the syntax of inference rules. A type rule  $r$  consists of zero or more premisses (judgements  $\bar{j}$ ) above the line, and a conclusion  $j$  below the line. We come back to meta expressions  $m$  later, but it contains at least the meta variables  $x$ .

Figure 1.4 shows the rules of the lambda calculus as introduced above. The rule CON associates the type  $I$  to any integer constant. The VAR rule associates the type to the identifier as it is bound to that identifier in the environment. The APP rule requires expression  $f$  to be a function that takes an argument of the same type as its formal parameter. The type associated with the application itself is the result type of the function. In LAM, the body of the lambda may assume a type  $\tau_1$  associated with identifier  $x$  in the environment.

The typing relation is the smallest relation that satisfies a set of rules  $r$ , which serve as axioms of the relation. In a judgment, meta expressions  $m$  are arguments to some relation  $R$ . The language of meta expressions is a formal language, which is usually left implicit, but facilitates the construction of symbolic and concrete object terms.

A Relation  $R$  can be a built-in relation (an *atomic* relation) or relation described by inference rules. Instead of using expressions  $m$  to construct symbolic terms, atomic relations for each production in the object language can equivalently be used to constrain symbolic object

terms. For example, for the arrow-production in the syntax of types, we assume the existence of an atomic relation  $C_{\rightarrow} \tau_1 \tau_2 \tau_3$  that expresses  $\tau_1 \equiv \tau_2 \rightarrow \tau_3$ . Then, instead of a judgement  $R(\tau_2 \rightarrow \tau_3)$ , we may write  $R \tau_1$  with a fresh  $\tau_1$  and  $C_{\rightarrow} \tau_1 \tau_2 \tau_3$ .

The rules are in canonical form when the arguments of relations in the judgments consist only of meta variables, with the single exception of the judgment denoted  $m \rightsquigarrow^* x$  which represents a reduction relation on object terms where  $m$  is a meta expression. Additionally, each meta variable occurs at most once, not counting its occurrences in (additional) explicit equalities. We assume in this thesis that meta expressions are written in Haskell. For reasons of simplification we assume below that meta expressions in this chapter are written in the above lambda calculus so that the definitions of  $m$  and  $e$  coincide. To rewrite rules into canonical form we use the operational semantics of (meta) expressions. For example, we replace a judgment  $R e$  to  $R x$  and introduce the additional premise that  $e \rightsquigarrow^* x$ , where  $x$  is fresh and  $\rightsquigarrow^*$  is the exhaustive application of the reduction relation on (meta) expressions.

To deal with the occurrences of meta variables, we introduce additional explicit equalities and quantified fresh variables. Rules in canonical form are more verbose, but easier to formally reason with.

Given arguments  $\bar{a}$  for the typing relation  $R$  described by inference rules, we can prove that these are a member of the typing relation, denoted by  $R \bar{a}$  or  $\bar{a} \in R$ , by constructing a *derivation tree* using the rules of the relation. A derivation tree is a proof for  $R \bar{a}$  when there is a rule of  $R$  with a conclusion that matches against  $R \bar{a}$  (with substitution  $\theta$ ), and for each premise of the rule (with  $\theta$  applied), there is subtree that is a proof for that premise. Proofs of atomic relations are leafs of a derivation. A derivation tree for the earlier example is:

$$\frac{\frac{\text{VAR} \frac{(y:I \rightarrow I) \in \Gamma_1}{\Gamma_1 \vdash y:I \rightarrow I} \quad \Gamma_1 \vdash 3:I \text{ CON}}{\Gamma_1 \vdash y 3:I} \text{ APP} \quad \frac{(x:I) \in \Gamma_2 \quad \text{VAR}}{\Gamma_2 \vdash x:I} \text{ VAR}}{\Gamma_0 \vdash \lambda(x:I).x:I \rightarrow I} \text{ LAM}}{\Gamma_0 \vdash (\lambda y:I \rightarrow I.f 3) (\lambda x:I.x):I} \text{ APP} \text{ LAM}$$

- $\Gamma_0 = \emptyset$                       -- initial environment
- $\Gamma_1 = \Gamma_0, y:I \rightarrow I$     -- extension of  $\Gamma_0$  in the application of LAM (left branch)
- $\Gamma_2 = \Gamma_0, x:I$                 -- extension of  $\Gamma_0$  in the application of LAM (right branch)

Each node  $v$  in the derivation tree is associated with a judgment  $j_v = R_v \bar{a}$ , and the subtree rooted by  $v$  is a proof that  $\bar{a} \in R_v$ , where  $R_v$  is the associated relation. Furthermore, node  $v$  is associated with some rule  $r_v$ , which determines the structure of  $v$ . The node is furthermore decorated with values for the meta variables that are bound by  $r_v$ .

Type rules are *syntax-directed* when in derivation trees, for each node  $v$ , the associated rule  $r_v$  is uniquely determined by an argument  $a$  (which represent the expression for which we try to construct the proof) of the associated judgment  $j_v$ . Thus, for syntax directed rules, the choice of what rule to apply in the construction of the proof is determined uniquely by productions of the object language, and there is a one-to-one mapping between nodes in the abstract syntax tree and nodes of the derivation tree. This is a desirable property because it

## 1 Introduction

means that the shape of the derivation tree depends only on the structure of the object term, and is thus given at the start of the proof construction: we know what rules to apply where, and are only left with the problem of how to make their instances match.

However, when the correspondence between  $r_v$  and the a priori known arguments  $\bar{i} \subseteq \bar{a}$  of  $j_v$  is not functional, we call the rules *declarative*<sup>3</sup>. Additionally, when  $r_v$  also depends on the inferred type  $\tau$ , with  $\tau \in \bar{a}$ , the rules are declarative and also *type directed*.

A rule  $r_v$  imposes two forms of constraints on node  $v$  in the derivation tree. Constraints on the structure (as discussed above), and constraints on the values for the node's meta variables that decorate  $v$  (as discussed below). When constructing a proof, we choose rules and values that satisfy these constraints. We call the description of such choices *aspects* of the rules. These aspects are declarative when the decisions are not functionally determined by  $\bar{i}$ . Declarative aspects complicate the construction of a proof, because there may be many choices that seem suitable to complement a partial proof, but turn out later to be inappropriate. In Section 1.2.4, we discuss strategies to resolve such declarative aspects.

We determine which meta variables are declaratively defined with a transformation of the relations into functions: those meta variables that make the function non-deterministic are declaratively defined. For that, each parameter of a relation must have a specification that declares it (conditionally) as either an input or output. In a judgement  $j = R \bar{x}$  of a rule in canonical form, a meta variable  $x \in \bar{x}$  is at an input position when  $j$  is a premise and  $x$  is passed as output argument, or when  $j$  is a conclusion and  $x$  is passed as an input argument. Otherwise, a meta variable is at an output position. In case of explicit equalities  $x_1 \approx x_2$ ,  $x_1$  and  $x_2$  are both at an input position. In case of the reduction relation  $m \rightsquigarrow x$ , the occurrences of meta variables of  $m$  are at an input position and  $x$  is at an output position.

In case of the construction/destruction relation  $C_P$  related to the production  $P$  of the object language, in the judgment  $C_P x_0 x_1 \dots x_n$ ,  $x_1, \dots, x_n$  are at an input position if and only if  $x_0$  is at an input position.

A variable is *declarative* if none of its occurrences are on an output position and the variable does not occur in an explicit equality with a non-declarative variable.

As an example, we add a type rule for an implicitly typed lambda abstraction, and discuss which meta variables are declarative. As preparation we show the canonical forms LAM-IMPL' and APP' in Figure 1.5. When we assume that types are an output of the relation, the meta variable  $\tau_1$  of LAM-IMPL is declarative. When we assume that types are an input of the relation, the meta variable  $\tau_1$  of APP is declarative.

The above sketch of an analysis assumes that the atomic relations are computable functions. We call type rules declarative if they exhibit declarative aspects, and *algorithmic* otherwise. In the latter case, when the rules are described in Ruler (Section 1.4), a type inference algorithm can be generated.

### 1.2.4 Type Inference

A type checking or type inferencing algorithm concerns itself with constructing derivation trees when given the main judgment. In case of type checking the type is part of these a

---

<sup>3</sup> Some authors call declarative rules *nondeterministic* because an inference algorithm may need to choose non-deterministically what rule to apply.



$$\begin{array}{c}
\frac{\Gamma, x: \tau_1 \vdash e: \tau_2}{\Gamma \vdash \lambda x. e: \tau_1 \rightarrow \tau_2} \text{LAM-IMPL} \qquad \frac{\Gamma \vdash f: \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash a: \tau_1}{\Gamma \vdash f a: \tau_2} \text{APP} \\
\\
\frac{C_{\rightarrow} m_2 \tau_1 \tau_2 \quad C_{App} m_1 x e}{\Gamma \vdash m_1: m_2} \text{LAM-IMPL}' \qquad \frac{C_{\rightarrow} m_2 \tau_1 \tau_2 \quad C_{App} m_1 f a}{\Gamma \vdash m_1: \tau_2} \text{APP}'
\end{array}$$

**Figure 1.5:** Rules LAM-IMPL and APP, and their respective canonical form.

priori known arguments. In case of type inference the type is not, and actually inferred. This distinction between type checking and type inference is rather vague, because a type system may exhibit other declarative aspects than the definition of types, and thus implies some form of inference. Moreover, for a compiler typically the inferred arguments matter but the derivation tree is not of direct interest.

**Inference as a forest of derivation trees.** Some authors consider type inference as a two-step process: a traversal of the AST to generate a set of constraints, and solving this set of constraints. We shall, however, consider type inference as the incremental construction of a forest of derivation trees, which has a closer connection to declarative specifications. Its intermediate state is a forest of partial derivation trees. It starts with a singleton forest containing an empty derivation tree with the main judgment as pending aspect. A derivation tree in the forest has an associated status, which is that it is partial, complete, or unsatisfiable.

A reduction step in this forest consists of cloning an existing partial derivation and adding the cloned derivation to the forest after resolving one pending aspect of the clone such that the number of unique derivation trees in the forest increases.

An aspect is either a meta variable or a judgment. To resolve the former, the meta variable is bound to a value in its domain. To resolve the latter, it depends on whether the relation of the judgment is described by rules or an atomic relation. If the relation is described by rules, either a rule can be applied which leads to a larger or a complete derivation tree, or the derivation tree is unsatisfiable. If the relation is atomic, then the aspect is solved by running an algorithm that is associated to the relation which either succeeds or fails.

The existence of a complete derivation means that the source program is type correct. When all reductions have been applied exhaustively and there exists no complete derivation the program is type incorrect. However, there may be infinitely many partial derivations. Also, there may be infinitely many partial derivations of a certain height, for example, when the domain of a meta variable is infinite. We come back to this issue later: in practice, an inference algorithm uses a less general approach.

**Naive algorithm.** The algorithm in Figure 1.6 represents a typical Prolog-style strategy for the construction of derivation trees. We explain some aspects of the example below.

## 1 Introduction

```
prove :: j → Inf Tree           -- Inf offers unification and backtracking
prove (x ≈ y) = do             -- case for explicit equality
  unify x y                    -- equality mapped to unification
  return (x ~ y)               -- builds equality node
prove (R x̄)                    -- case for judgments
  | atomic R = extern R x̄     -- proven externally
  | otherwise = do             -- relation specified by rules
    r ← rules R                -- picks a rule (backtrack point)
    (ps ; c) ← instantiate r   -- instantiate rule
    zipWithM unify (args c) x̄ -- bind args
    ts ← mapM prove ps         -- recursion on premises
    return (ts ▷name r c)      -- build derivation node
```

**Figure 1.6:** Sketch of a general inference algorithm.

In this example, we refer to an inference monad *Inf* which takes care of backtracking and unification. Rules are tried in a predefined order. If a rule cannot be applied, *backtracking* occurs to the next rule.

Meta variables are initially represented as symbolic values, and unified with other meta variables or object terms during inference. Unification is derived from an equivalence relation, which can in turn be derived generically from algebraic data type declarations.

The operation *rules* introduces a backtrack-point. It tries the rules in a given order by feeding the rules one by one to the continuation. A later unification may fail and cause a backtracking to that point. The operation *instantiate* substitutes the quantified meta variables with fresh meta variables. The instruction *extern* proves a judgment externally and returns evidence for it if it succeeds. With combinators  $\triangleright$  (internal nodes) and  $\sim$  (equality leaves) we construct evidence in the form of a derivation tree.

**Undesirable properties of the algorithm.** The above algorithm is incomplete and does not terminate for all but the simplest type systems. This is necessarily the case because there exist type systems for which inference is undecidable. Via backtracking, only finite and inductively defined object terms can be inferred, and unifications only produce compositions of object terms. The algorithm is also inefficient. The order in which rules are tried may cause poor performance or nontermination. Moreover, the order in which rules are tried is not specified, which gives unpredictable results. Although the results are sound, these may not be optimal.

In practice, typical algorithms refrain from backtracking or constructing many candidate derivations, because the search space is too large when dealing with ASTs with thousands of nodes. Therefore, actual inference algorithms resemble this overall approach, but select rules and instantiations of meta variables in a more sophisticated way.

**Actual inference algorithms.** Actual inference algorithms apply various strategies (depending on the form of the declarative aspect) to get around the above undesirable properties.

Unification can be used to deal with equivalence judgments in type rules. Alternatively, an algorithm may collect several candidate constraints on a meta variable, then pick an object term that is the least solution to all these constraints. When constraints are monotonic, a value can gradually be approximated via fixpoint iteration with an initial bottom value. Type and effect systems often require such algorithms.

When rule selection is declarative, we typically want the choice to be a function of the syntax of the language (*syntax directed* rules) or other object terms (e.g. *type directed* rules). Alternatively, some rules can be restricted to only be applied in a proof after other rules have been applied so that the choices between the remaining rules becomes functional in the above sense. This is not always possible: the applicability of a rule may depend on unresolved meta variables and may require the rule selection to be deferred which is called residuation Hanus [1994].

**Challenge: orchestration of strategies.** The orchestration of such strategies is a complex undertaking. The order in which strategies are applied may influence the result, and it may not always be clear when to start or stop applying strategies. These are all challenges an implementation must deal with.

**Challenge: annotations.** The holy grail of type system research is to define expressive type systems (for a class of programs) that have sound and complete (decidable) inference algorithms. Given two type systems (that satisfy type-soundness with respect to the operational semantics of the language), one type system is more expressive than another type system if it accepts a superset of the programs that the other accepts.

To bypass the strict undecidability boundaries, many languages allow programmers to assist the inference process by providing additional information (e.g. type annotations) in the object program that translates to concrete bindings for otherwise declarative meta variables. There are delicate balances between expressiveness of the type system, the amount of annotation to be provided by the programmer, and the predictability of inference.

For example, the Damas-Hindley-Milner type system and accompanying inference algorithm (Section 1.2.6) does not require any annotations to help the inference process, but disallows functions with parameters of a polymorphic type. System F, the polymorphic lambda calculus, in comparison allows functions with a polymorphic type, but requires an abundance of type annotations. As a middle way, HML [Leijen, 2009] expresses all of System F, but requires only type annotations for polymorphic lambda parameters. FPH [Vytiniotis et al., 2008] positions itself in between DHM and HML. It requires a type annotation when applying a function to an argument with a polymorphic type. Dijkstra and Swierstra [2006a] proposed a global flow analysis that propagates type annotations to locations where (a part of) the annotation is also applicable.

**Challenge: type errors.** As stated before, type inference concerns itself with the construction of derivation trees. If no such derivation exists, it is considered a type error. Type rules

## 1 Introduction

only specify under what condition a type is acceptable. Thus, an implementation may require additional information to produce understandable error messages [Heeren et al., 2003a].

**Challenge: multiple derivation trees.** A partial order on types specifies when one type is more general than another type. A type is *principal* when it is the most general type of an expression. For reasons of predictability and modularity it is desirable that a principal type exists for an expression and that an algorithm infers a derivation for this type. This notion of principality can be extended to derivations [Jim, 1996].

For the above type rules, there are infinitely many derivation trees for  $(\lambda x.x)$ , but none of the accompanying types (e.g.  $\tau \rightarrow \tau$  for any type  $\tau$ ) are comparable, thus this expression has no principal type (thus also not a principal derivation) for the given type system.

There are several remedies. The type system can be changed such that more type annotations need to be given, or that choices can be deferred by encoding pending choices in the type language as constraints. Polymorphism and qualified types are an example of the latter: type schemes are introduced to represent a delayed choice of types, as we show in the next section.

### 1.2.5 Parametric Polymorphism

For the identity-function  $\lambda x.x$ , there are no constraints on the type of  $x$ . During inference, no binding arises for the meta variable associated to  $x$ . We can thus bind a fresh type constant  $\alpha$  to the meta variable, which leads to the type  $\alpha \rightarrow \alpha$  for  $\lambda x.x$ . To specify that  $\alpha$  can be any type, we universally quantify over  $\alpha$ , and obtain the type  $\forall \alpha. \alpha \rightarrow \alpha$ . This process is called abstraction or *generalization*. A polymorphic type represents many types which can be obtained by *instantiating* the quantified type constants (*type variables*). Moreover, we obtain a proof for the instantiated type by instantiating the proof for the generalized type in an analogous way.

Polymorphic types enable *parametric polymorphism*, which allows functions to be called with parameters of different but acceptable types. Parametric polymorphism is important for the use of Haskell's many convenient higher-order functions, such as *id*, *flip* and *\$*.

**Quantified types.** The *Damas-Hindley-Milner* (DHM) type system serves as the classical example of a type system with polymorphic types. It forms the basis of the type systems of ML and Haskell, and underlies the type systems of many other languages. DHM's type language is an extension to that of the simply typed language calculus. It additionally contains type variables  $\alpha$  and poly types (*type schemes*)  $\sigma$ :

$\tau ::= \alpha$	-- type variable
$\tau_1 \rightarrow \tau_2$	-- function type
$\sigma ::= \forall \alpha. \sigma$	-- poly type: may have a qualifier
$\tau$	-- mono type: has no quantifiers
$\alpha, \beta$	-- type variables

$$\boxed{
\begin{array}{c}
\Gamma \vdash e : \sigma \\
\\
\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma} \text{VAR} \qquad \frac{\Gamma \vdash f : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash a : \tau_1}{\Gamma \vdash f a : \tau_2} \text{APP} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \text{LAM} \\
\\
\frac{\Gamma \vdash e : \sigma \quad \Gamma, x : \sigma \vdash b : \tau}{\Gamma \vdash \mathbf{let } x = e \mathbf{ in } b : \tau} \text{LET} \qquad \frac{\Gamma \vdash e : \sigma \quad \alpha \notin \text{fv } \Gamma}{\Gamma \vdash e : \forall \alpha. \sigma} \text{GEN} \qquad \frac{\Gamma \vdash e : \forall \alpha. \sigma}{\Gamma \vdash e : \sigma [\alpha := \tau]} \text{INST}
\end{array}
}$$

Figure 1.7: The DHM type system.

A mono type  $\tau$  does not contain universal quantifiers. A poly type can be interpreted as the infinite set of mono types, where each quantified type is substituted by some mono type. The identity function  $\lambda x. x$  has the type  $\tau \rightarrow \tau$  for any type  $\tau$ . It can thus be given the poly type  $\forall \alpha. \alpha \rightarrow \alpha$ . The type variable  $\alpha$  in such a type is an object term that is not to be confused with a meta variable.

The distinction between mono and poly types is important in the declarative type rules of DHM. Lambda abstractions take mono types as parameter<sup>4</sup>, and have a mono type as result. Polymorphic types are introduced in the environment by (non-recursive) let-expressions:

$$\begin{array}{l}
e ::= \dots \\
| \mathbf{let } x = e_1 \mathbf{ in } e_2 \quad \text{-- generalized type of } e_1 \text{ is visible as } x \text{ in } e_2
\end{array}$$

Although a let-expression can be interpreted as  $(\lambda x. e_2) e_1$ , the typing derivation may differ depending on the presence and application of a generalization rule.

The DHM type system in Figure 1.7 consists of the following rules of which we mention some details below. Via one or more applications of the INST rule, poly types may be instantiated to mono types by replacing a bound type variable with a mono type. Conversely, via rule GEN a poly type can be constructed using quantification over a type variable, provided that the substitution does not capture the free type variables in the environment.

**Qualified types.** In general, meta variables may be constrained by judgments that by themselves are insufficient to bind a concrete type to a meta variable. As an example, the following type rules encode overloading of the addition operator on both integers and floats.

$$\Gamma \vdash \_ + \_ : I \rightarrow I \rightarrow I \text{ ADD.INT} \qquad \Gamma \vdash \_ + \_ : F \rightarrow F \rightarrow F \text{ ADD.FLOAT}$$

<sup>4</sup> It has been shown that the inference of a polymorphic type for a parameter of a recursive function is undecidable in general [Wells, 1999]. Many type systems thus impose restrictions on the types of function parameters to make inference feasible.

## 1 Introduction

In the expression  $\lambda x.x + x$ , the type rules for addition constrain  $x$  to be a numeric type, but do not dictate whether this type is an integer or a float. At other locations in the derivation tree there may be constraints imposed on the type that make it clear which of the rules applies.

During type inference, the type may be insufficiently constrained to resolve the judgment. A typical strategy is to defer the judgment until the end of the inference of the scope in which the judgment arose, which is usually at generalization points. If the type is still not constrained sufficiently, a strategy is to *default* to one of the applicable rules. Another strategy is to encode the judgment as part of the type (if potentially satisfiable) and delay the judgment to all locations where the generalized type is instantiated. Such an encoded judgment is called a qualifier.

A qualified type of the expression  $\lambda x.x + x$  given the above rules is:

$$(\lambda x.x + x) :: \forall \alpha \beta. \exists \gamma. (\gamma \vdash \_ + \_ : \alpha \rightarrow \alpha \rightarrow \beta) \Rightarrow \alpha \rightarrow \beta$$

With additional information this type can be refined. An equivalence between  $\beta$  and  $\alpha$  is deducable from the rules `ADD.INT` and `ADD.FLOAT`. Also, suppose that we know further that a qualifier  $\gamma \vdash \_ + \_ : \alpha \rightarrow \alpha \rightarrow \alpha$  is simplifiable to a qualifier `Num`  $\alpha$ , the type is refineable to:

$$(\lambda x.x + x) :: \forall a. \text{Num } a \Rightarrow a \rightarrow a$$

Haskell's overloading with type classes actually brings such reasoning under the control of the programmer.

Such extensions to a type system do not come for free. When the code generation depends on the proof of a deferred judgment, the generated code needs to be parametrized by information that is derived from the deferred proof. Moreover, a function may be given a type with qualifiers that can never be satisfied, which we may only find out when we try to use an identifier with such a type. Some type systems define a coherence relation on qualifiers to formalize the potential satisfaction of constraints.

### 1.2.6 Damas-Hindley-Milner Inference

In this section we consider type inference for the DHM type system. The naive backtracking approach as presented earlier may easily lead to nontermination, because the `INST` and `GEN` rule can be alternated as each others inverse indefinitely. However, the syntactical restrictions on types permits a more appropriate inference strategy.

With some effort, we can deduce that generalization has only an effect for the toplevel expression, or for the expression  $e$  in a let-expression. It is also sufficient to perform instantiation only after taking the type of an identifier from the environment in the rule `VAR`. According to the interpretation of poly types, a type quantified over a variable that does not occur free in the type describes actually the same set of mono types as the type without the quantification, thus we only need to consider types that occur free in the type as generalization candidates. Moreover, the order of the type variables over which is quantified is irrelevant.

Inference algorithm `W` exploits these properties. It is a sound and complete implementation of the rules, produces most general types, and needs to examine each node of the AST only

In the description of algorithm W we assume monadic versions of the above functions to form an inference monad  $Inf$  for the implementation of the DHM type system. Essentially, these monadic functions are wrappers around the above functions:

```

type  $Inf = RWST\ Env\ Subst\ Errs$   -- reader, writer, and state monad
 $fresh_I :: Ty \rightarrow Inf\ Ty$           -- returns a fresh type
 $unify_I :: Ty \rightarrow Ty \rightarrow Inf\ ()$    -- unifies two types
 $gen_I :: Ty \rightarrow Inf\ Scheme$       -- generalizes a type to a scheme
 $inst_I :: Scheme \rightarrow Inf\ Ty$       -- instantiates a scheme to a type

```

The threading of the substitution and collection of error messages is hidden in the monad, as well as the top-down distribution of the environment, so that we define algorithm W as a function  $\mathcal{W} :: Expr \rightarrow Inf\ Ty$ :

```

 $\mathcal{W}\langle x \rangle = asks\ (lookup\ x) \gg\gg\ inst_I$ 
 $\mathcal{W}\langle \lambda x.e \rangle = do$ 
   $t \leftarrow fresh_I$ 
   $r \leftarrow local\ (insert\ x\ t)\ (\mathcal{W}\ e)$ 
   $return\langle t \rightarrow r \rangle$ 
 $\mathcal{W}\langle f\ a \rangle = do$ 
   $r \leftarrow fresh_I$ 
   $t \leftarrow \mathcal{W}\ f$ 
   $s \leftarrow \mathcal{W}\ a$ 
   $unify_I\ t\langle s \rightarrow r \rangle$ 
   $return\ r$ 
 $\mathcal{W}\langle let\ x = e\ in\ b \rangle = do$ 
   $t \leftarrow \mathcal{W}\ e \gg\gg\ gen_I$ 
   $local\ (insert\ x\ t)\ (\mathcal{W}\ b)$ 

```

The notation  $\langle e \rangle$  represents the abstract syntax of an object term  $e$ . We use this notation to conveniently pattern match and construct object terms in the host language. Note that  $\langle x \rangle$  is an AST of the type  $Expr$  and  $x$  is an AST of the type  $Identifier$ , but that  $\langle f\ a \rangle, f$  and  $a$  represent ASTs of the type  $Expr$ .

**Figure 1.8:** Algorithm W.

once. Many actual inference algorithms are based on algorithm W. We describe algorithm W as a monadic function  $\mathcal{W} :: Expr \rightarrow Inf\ Ty$  in Figure 1.8.

Unification plays an important role in algorithm W. In this thesis, we assume the following functions have an efficient implementation [Dijkstra et al., 2008]:

```

 $fresh :: Subst \rightarrow (Ty, Subst)$   -- returns a fresh type var
 $unify :: Ty \rightarrow Ty \rightarrow Subst \rightarrow (Errs, Subst)$   -- unify two types (improves subst)
 $generalize :: Env \rightarrow Ty \rightarrow Subst \rightarrow Scheme$   -- generalize a type in a certain env
 $instantiate :: Scheme \rightarrow Subst \rightarrow (Ty, Subst)$   -- instantiate a scheme freshly

```

The type  $Subst$  represents a substitution and a fresh variable supply, which is a mapping from meta variable to a concrete type. The types  $Ty$  and  $Scheme$  coincide with  $\tau$  and  $\sigma$  respectively. An environment of type  $Env$  contains bindings from identifiers to types, and  $Errs$  is the type of a collection of error messages. Similarly, we assume that there is a type  $Expr$  and  $Identifier$  that coincide with the nonterminals  $e$  and  $x$  respectively.

Figure 1.8 shows Algorithm W, which is a recursive traversal of the  $Expr$  AST. The combi-

$\mathcal{W} \langle \mathbf{let} \ x = e \ \mathbf{in} \ b \rangle = \mathbf{do}$	
$t \leftarrow \mathit{fresh}_I$	-- start with fresh type for binding
$\mathit{local} \ (\mathit{insert} \ x \ t) \ (\mathcal{W} \ e \gg \gg \mathit{unify}_I \ t)$	-- infer type and match against binding
$s \leftarrow \mathit{gen}_I \ t$	-- generalize over unbound variables
$\mathit{local} \ (\mathit{insert} \ x \ s) \ (\mathcal{W} \ b)$	-- infer with generalized scheme in env

**Figure 1.9:** Algorithm W with a recursive let-binding.

nator *local* applies changes to the environment that are only visible in the monadic computation it encapsulates. The function *insert* adds a binding into the environment, which possibly shadows an already existing binding. The *asks* operator exposes the hidden environment, so that we can use *lookup* to recover the type to which an identifier is bound.

The above code assumes a non-recursive let binding. Figure 1.9 shows the algorithm for a recursive let binding. The unification at the end of inference for *e* binds *t* to its actual type. All occurrences of *x* in its own right-hand side must agree with *t*. To prevent having to deal with polymorphic recursion, *x* has a mono-type inside its own binding. The generalized type is only available in the body of the let-expression.

The positioning of *local* and *gen<sub>I</sub>* is tricky. Since generalization is performed with respect to free type variables in the environment, the generalization needs to be positioned outside the local environments of the *e* and *b* subexpressions, because the type to generalize occurs in these environments.

The monadic formulation of algorithm W is concise because cumbersome flows of environments and substitutions are encapsulated by the monad. However, the abstraction offered by the monad is not always obvious when the object language is more complex. When the object language has pattern-bindings, new bindings arise when visiting the pattern, which means that behavior for the environment is more complex than simply top-down. If multiple derivations are possible, then substitutions may need to be duplicated and merged. The sequencing of operations on the encapsulated state is therefore important.

## 1.2.7 Polymorphic Lambda Calculus

When we eliminate the distinction between mono and poly types, and thus allow poly types everywhere, we obtain an implicitly typed version of the *System F* type system. It allows expressions such as  $\lambda f.g \ (f \ (\lambda x.x)) \ (f \ 3)$ , where a lambda parameter is applied to values of different types, which is not expressible in the DHM type system. Unfortunately, inference for this system is undecidable [Wells, 1999], and it does not have most general types.

For System F itself, type checking is decidable. However, the syntax is verbose, as type abstraction and type instantiation are explicitly encoded, and types need to be given explicitly for lambda parameters. Figure 1.10 shows the inference rules of System F, and we discuss some aspects of the rules below.

A type application  $f \ \sigma$  requires *f* to have a universally quantified type, and provides the



$$\boxed{
\begin{array}{c}
\Gamma \vdash e : \sigma \\
\\
\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma} \text{VAR} \qquad \frac{\Gamma \vdash f : \sigma_1 \rightarrow \sigma_2 \quad \Gamma \vdash a : \sigma_1}{\Gamma \vdash f a : \sigma_2} \text{APP.E} \qquad \frac{\Gamma \vdash f : \forall \alpha. \sigma_2}{\Gamma \vdash f \sigma_1 : \sigma_2 [\alpha := \sigma_1]} \text{APP.TY} \\
\\
\frac{\Gamma, x : \sigma_1 \vdash e : \sigma_2}{\Gamma \vdash \lambda x. e : \sigma_1 \rightarrow \sigma_2} \text{ABS.E} \qquad \frac{\Gamma \vdash e : \sigma \quad \alpha \notin \text{fv } \Gamma}{\Gamma \vdash \Lambda \alpha. e : \forall \alpha. \sigma} \text{ABS.TY}
\end{array}
}$$

Figure 1.10: System F.

type  $\sigma$  to instantiate this type with. A type abstraction  $\Lambda \alpha. e$  quantifies the type of  $e$  over  $\alpha$ . The syntax of  $f$  thus dictates when to apply rule `APP.TY` and rule `ABS.TY`.

Variants of System F are typically used for typed backends of compilers, because of its expressiveness. For example, evidence translation of Haskell type classes may need System F types [Faxén, 2002]. As frontend, type systems are proposed that are more restrictive than System F, but more liberal than the DHM type system. It is desirable that such systems do not require type annotations for programs that are acceptable by DHM, and allow for a predictable inference. An inference algorithm then infers a type for such programs, and maps these to System F terms.

### 1.2.8 Discussion

The DHM inference algorithm in Section 1.2.6 shows that the mapping between a declarative type system specification and its accompanying inference algorithm is not straightforward. The gap between a complex type system and a sophisticated inference algorithm is even larger. As mentioned above, one of reasons is that an inference algorithm requires information that is not present in type rules. Also, type derivations are typically only treated as a model. Inference algorithms compute types from abstract syntax trees, with the underlying assumption that the combination of type and AST can be turned into a derivation tree.

Generalization and instantiation, for example, are often dealt with as part of the unification algorithm in order to support on-demand *impredicative instantiation*, which is the instantiation of a type variable with a poly type. In the type rules these are encoded as separate expression rules.

A declarative specification is typically a minimalistic lambda calculus to explain particular language features, whereas programming languages are much richer in syntax and language features. Algebraic data types are typically not present in declarative specifications, and neither are mutually recursive let bindings, because these are often regarded as syntactic sugar. Actual programming languages, however, require the presence of such features.

We can improve the resemblance between the inference algorithm and declarative specification when the structure of the inference algorithm can be derived from the declarative

specification. In this thesis, we propose the use of attribute grammars as the basis for such algorithms. Section 1.3 gives a short introduction.

### 1.3 Background on Attribute Grammars

In Section 1.2, we followed the common practice of specifying the semantics of programming languages via relations between properties, where the relations are defined by inference rules and the properties include a source term. *Attribute Grammars* (AGs) [Knuth, 1968] are an alternative approach. An AG specifies the semantics of a language as attributes on nonterminal symbols in the grammar of the language.

An AG is a context-free grammar extended with *attributes* and *rules*. We give a definition in Section 1.3.1, in which we describe that attributes are associated with nonterminals, and rules with productions. Given a value for each attribute associated with the nonterminal symbols of a production, the rules specify whether these values are correct. On the other hand, the rules can also be used to compute such attributions for a given AST, which we address in Section 1.3.4.

In the other subsections we describe common extensions, features and uses of attribute grammars for the purpose of showing why AGs are an attractive language for describing the implementation of compilers, but also to give some background information to which we refer from later chapters of this thesis. A shallow scan through these subsections may be beneficial to the understanding of the other chapters of this thesis.

Section 1.7.6 gives an overview of various systems that provide AGs in various flavors.

#### 1.3.1 Syntax of Context-Free Grammars and Attribute Grammars

We introduce a notation for attribute grammars. In this section, we use the term *host language* to refer to the language in which the algorithm or compiler is written which we generate from the attribute grammar description. As we see later, functions of the host language may appear in grammar descriptions.

We are slightly more formal in this section in comparison to the other sections because in later chapters we introduce various notations for attribute grammars and extensions. Since concepts such as nonterminals, productions, attributes and abstract syntax trees are common to those notations, we introduce these here — although we actually expect that the reader is already familiar with these definitions.

**Context-free grammars.** Formally, grammars specify languages, but we also use grammars to describe the structure of tree-like data structures. Chomsky [1956] described several classes of grammars with increasing expressiveness and implementation complexity. The class of context-free grammars is particularly convenient for the description of the structure of terms, but is usually not expressive enough to describe the desired correctness properties of programs such as well-typedness. With attribute grammars, context-free grammars are combined with a different formalism to permit the description of such properties.

**Definition** (Context-free grammar). A *context-free grammar* is a tuple  $(V, N, S, P)$  where  $V$  is a set of terminal symbols (the *alphabet*),  $N$  is a set of nonterminal symbols,  $S$  is the start symbol with  $S \in N$ , and  $P$  is a set of productions (defined below). The set  $V$  and  $N$  must be disjoint.

**Definition** (Production). A (context-free) *production*  $p = n \rightarrow \bar{m}$  is a rewrite rule with nonterminal symbol  $n \in N$ , and a sequence of symbols  $\bar{m} \in V \cup N$ . The sequence  $\bar{m}$  may be empty. The application of  $p$  to a sequence of symbols  $\bar{s} \in V \cup N$  constitutes to the rewriting of one occurrence of  $n$  in  $\bar{s}$  to  $\bar{m}$ . In production  $p$ ,  $n$  forms the left-hand side of the production and  $\bar{m}$  the right-hand side.

To summarize, a grammar is a rewriting system where productions specify a rewrite step from a sequence of symbols to sequence of symbols. In a context-free grammar, a production specifies how to rewrite a single nonterminal symbol to a sequence of symbols. The rewriting terminates when only terminal symbols are left: when successive applications of productions to some singleton sequence  $n$  (with  $n \in N$ ) results in a sequence of symbols, then this sequence is derived from  $n$ .

**Definition** (Sentence). A *string* is a sequence of symbols. A *sentential form* is a string derivable from the start symbol of the grammar. A *sentence* is a sentential form consisting only of terminal symbols.

**Definition** (Derivation tree). A derivation tree is a tree  $t$  that represents how a sentence  $\bar{s}$  is derived from a symbol  $m$ , and is inductively defined as follows:

- A leaf  $t$  represents either the derivation of the empty string from a nonterminal symbol  $n$  if there exists a production  $p = n \rightarrow \varepsilon$ , or the trivial derivation of the singleton string  $\bar{s} = v$  from a terminal symbol  $v$ . (Only) in the former case, we say that the leaf is associated with the nonterminal  $n$  and the production  $p$ . In the latter case, the leaf is only associated with the terminal  $v$ .
- If trees  $t_1, \dots, t_k$  represent the respective derivations of sentences  $\bar{s}_1, \dots, \bar{s}_k$  from symbols  $m_1, \dots, m_k$  then the tree  $t$ , formed by taking  $t_1, \dots, t_k$  as the respective children of the root, represents the derivation of the sentence  $\bar{s}_1 \dots \bar{s}_k$  from symbol  $n$  if there exists a production  $p = n \rightarrow m_1 \dots m_k$ . We say that the root of  $t$  is associated with the nonterminal  $n$  and the production  $p$ .

**Definition** (Syntax tree). A *syntax tree* (or *parse tree*) is a derivation tree that is associated with the nonterminal  $n$ . This definition purposefully excludes singleton trees denoting a terminal symbol.

**Definition** (Abstract syntax tree). An abstract syntax tree is the result of applying a projection to some syntax tree. Usually, the resulting tree has less branches (e.g. due to omission of layout) or branches replaced with a more general representation (e.g. desugared).

In this thesis, we do not concern ourselves with parsing, which is the process of constructing a syntax tree that represents the derivation of a given sentence. Instead, we assume the syntax tree as a given. In fact, we adopt the common convention to abstract over details in the syntax tree (e.g. whitespace) and work with abstract syntax trees instead.

$g ::= \mathbf{grammar} \ N \ \bar{p}$	-- the grammar for nonterminal $N$
$p ::= \mathbf{prod} \ P \ \bar{s}$	-- production $P$ , with as RHS the sequence of symbols $\bar{s}$
$s ::= \mathbf{term} \ x :: \tau$	-- a terminal symbol with name $x$ and type $\tau$
$\mathbf{nonterm} \ x : N$	-- a nonterminal symbol with name $x$ and nonterminal $N$
$N, P, x$	-- names of nonterminals, productions, and symbols

**Figure 1.11:** Language to describe context-free grammars.

**Definition** (Language). The *language*  $L_G$  specified by a grammar  $G$  is the set of sentences that can be derived from the start symbol of  $G$ .

If  $L_G$  is a programming language, then the source code of a program written in  $L_G$  is a sentence in  $L$ . Moreover, if  $L_G$  is a language of algebraic data types, then nonterminals describe type constructors, terminals describe primitive types, and productions describe data constructors. An AST represents a data structure, and the bit sequence in memory can be regarded as the sentence.

**Context-free grammar notation.** In Figure 1.11 we introduce a language for the *description* of context-free grammars: i.e. terms in this language can be interpreted as a grammar as defined above. We later extend the language to describe attribute grammars and some AG extensions. Figure 1.12 shows an example. We explain some aspects of the notation below.

**Definition** (Meta grammar). When we talk about a grammar for a grammar, we call the former a *meta grammar*.

In the notation, a grammar is the composition of grammars for individual nonterminals. The set of terminals  $V$  and nonterminals  $N$  are left implicit, and productions  $P$  are grouped per nonterminal. We reuse these letters for other purposes, such as  $N$  as an identifier for a nonterminal, and  $P$  as an identifier for a production.

The grammar is abstract: instead of terminal symbols, only the *type* of a terminal symbol is given. To stress the difference between terminals and nonterminals, we use a double colon to specify the type of a terminal and a single colon to specify the name of a nonterminal.

**Definition** (Children). Each symbol in the right-hand side of a production has an explicit name, which will be useful later. We call such named symbols the *children* of the production, which stresses the correspondence to children in the AST of nodes to which the production is associated.

In the notation, the nonterminal symbol of the left-hand side of a production is implicit, since we only describe the right-hand sides of productions. We give the symbol on the left-hand side of a production the fixed name *lhs*.

<b>grammar</b> <i>String</i>	-- a cons-list of characters
<b>prod</b> <i>Nil</i>	-- empty list
<b>prod</b> <i>Cons term</i> <i>hd</i> :: <i>Char</i>	-- head of the list ( <i>hd</i> is a terminal)
<i>tl</i> :: <i>String</i>	-- remainder ( <i>tl</i> is a nonterminal)

**Figure 1.12:** Example of a context-free grammar.

**Attribute grammars.** We now extend the above formalism to denote context-free grammars with notation for attributes and their associated functions.

**Definition** (Attribute grammar). An attribute grammar is a tuple  $(T, N, S, A, I, O, P, F)$ , where the set of terminals  $T$ , set of nonterminals  $N$ , and start symbol  $S$  are defined as for a context-free grammar. The set  $A$  consists of attribute names. The map  $I$  associates a set of names  $I_n \subseteq A$  with each nonterminal  $n$  in  $N$ , which make up the *inherited* attributes of  $n$ . Similarly, the map  $O$  associates a set  $O_n \subseteq A$  with each nonterminal  $n$  in  $N$ , which makes up the *synthesized* attributes of  $n$ . For each  $n$ , the sets  $I_n$  and  $O_n$  must be disjoint. The productions  $p \in P$  are redefined below. The set  $F$  consists of computable functions  $f$  which we call *semantic functions*.

**Definition** (Production). An (attribute-grammar) production  $p = u \rightarrow \bar{w} \cdot \bar{r} \cdot X$  consists of an annotated nonterminal symbol  $u$  and annotated symbols  $\bar{w}$ , rules  $\bar{r}$ , and a set of symbol names  $X$ .

**Definition** (Annotated symbol). An *annotated symbol* is either an annotated terminal or non-terminal symbol. An *annotated nonterminal symbol*  $u = x : n \cdot \bar{a}$  is a combination of a distinct symbol name  $x \in X$ , a nonterminal symbol  $n \in N$ , and a collection of attribute names  $\bar{a} \in A$  so that  $a$  is either in  $I_n$  or  $O_n$ . We say that  $n$  is associated to  $x$ . An *annotated terminal symbol*  $x : v$  is a combination of a distinct symbol name  $x \in X$  and a symbol  $v \in V$ .

**Definition** (Attribute occurrence). A *reference to an attribute*  $x.a$  is a combination of a symbol name  $x \in X$  (associated to some nonterminal  $n$ ) with an attribute name  $a \in A$  so that either  $a \in I_n$  or  $a \in O_n$ . A *reference to a terminal*  $x$  is a symbol name  $x \in X$  which is associated to some terminal  $v$ . An *attribute occurrence*  $o$  is either a reference to an attribute or a reference to a terminal.

We call an occurrence  $x.a$  also an attribute  $a$  of  $x$ . Attribute occurrences can be found in rules, which are defined below.

**Definition** (Rule). A rule  $\bar{o}_1 = f \bar{o}_2$  of some production  $u \rightarrow \bar{w} \cdot \bar{r} \cdot X$  consists of a semantic function  $f \in F$  and attribute occurrences  $\bar{o}_1$  and  $\bar{o}_2$ .

The occurrences  $\bar{o}_1$  represent the attributes defined by the rule, which are synthesized attributes of  $u$  or inherited attributes of the children  $\bar{w}$ . The occurrences  $\bar{o}_2$  represent the attributes used by the rule, which are the inherited attributes of  $u$  or synthesized attributes of the children  $\bar{w}$ . In addition, occurrences in  $\bar{o}_2$  may also refer to terminals.

## 1 Introduction

Due to the restrictions on occurrences it is always clear whether an occurrence references an inherited or a synthesized attribute. In our notation for attribute grammars (further below) we allow the same name to be used for an inherited and a synthesized attribute.

**Decorated trees.** To give a semantics to rules, we consider syntax trees annotated with attributes, and define how attributes of the tree are related to attribute occurrences, which are mentioned in the rules of productions.

**Definition** (Attributes of syntax trees). An *annotated syntax tree* or *semantic tree* is a syntax tree  $T$  where in addition subtrees are associated with the smallest set  $Q$  defined as follows. Let  $t$  be a subtree and  $n$  be the nonterminal that is associated to  $t$ . For each attribute  $a$  in  $I_n \cup O_n$ , let there be a distinct attribute symbol  $q_t \in Q$ . The set  $Q$  represents the attributes of  $T$ .

The symbols  $q$  can be seen as instances of the attributes, or as occurrences of the attributes in the tree. This definition states that many trees may be associated with the same nonterminal yet have different instances of the attributes. Moreover, if  $t$  is an annotated syntax tree, then the attributes of each annotated direct subtree of  $t$  are a distinct subset of the attributes of  $t$ .

**Definition** (Attribute association). Given some annotated syntax tree  $t$  with associated production  $p$ , an *attribute association*  $\alpha$  is a mapping so that for each attribute occurrence  $o$  of  $p$ , either  $\alpha o = q$  for some  $q \in Q$  when  $o$  is a reference to an attribute (either of  $t$  or of a direct subtree of  $t$ ), or  $\alpha o = v$  when  $o$  is a reference to a terminal child  $v$  of  $t$ .

For each (node of an) annotated syntax tree, there exists such an attribute association. We leave open how this straightforward connection between attribute occurrences and attributes of the syntax tree is constructed.

**Definition** (Valuation). A *valuation*  $M$  is a mapping that associates with each  $q \in Q$  and each  $v \in V$  a value in the host language, which is denoted as  $M q$  or  $M v$ . Furthermore,  $M v = \llbracket v \rrbracket$  for  $v \in V$  where  $\llbracket v \rrbracket$  is some encoding of  $v$  as a value in the host language. These values are called *decorations*.

**Definition** (Attributed syntax tree). An *attributed* (or *decorated*) syntax tree is an annotated syntax tree combined with a valuation  $M$ .

A rule  $\bar{o}_1 = f \bar{o}_2$  encodes the condition  $\overline{M(\alpha o_1)} = f \overline{M(\alpha o_2)}$  for each node the rule is associated with. Alternatively, we may say that  $f$  functionally defines occurrences  $o_1$  in terms of occurrences  $o_2$ , and thus that inherited attributes of children are defined by the parent, whereas synthesized attributes of the children are defined by the children and may be used by the parent.

**Definition** (Correctly attributed syntax tree). A syntax tree is *correctly attributed* when the conditions imposed by the rules are satisfied.

$I ::= \mathbf{attr} N \bar{i} \bar{s}$	-- attribute declaration for nonterminal $N$
$i ::= \mathbf{inh} y :: \tau$	-- declares inherited attribute $y$ of type $\tau$
$s ::= \mathbf{syn} y :: \tau$	-- declares synthesized attribute $y$ of type $\tau$
$S ::= \mathbf{sem} N \bar{c}$	-- semantics definition for $N$
$c ::= \mathbf{prod} P \bar{r}$	-- semantics in the form of rules for prod $P$
$r ::= g [\bar{o}_1] = f [\bar{o}_2]$	-- a rule with pattern $g$ and function $f$
$o ::= t.x.y$	-- attribute occurrence of child $x$ and attribute $y$ , and kind $t$
$t ::= \mathbf{inh} \mid \mathbf{syn} \mid \mathbf{loc}$	-- explicit attribute kind (usually left implicit)
$x, y$	-- names of symbols and attributes
$f, g$	-- expressions

Figure 1.13: Minimalistic language for AGs.

<b>attr</b> <i>String</i>	-- declares attributes of <i>String</i>
<b>inh</b> <i>down</i> :: <i>Char</i>	-- an inherited attribute <i>down</i>
<b>syn</b> <i>sh</i> :: <i>String</i>	-- a synthesized attribute <i>sh</i>
<b>sem</b> <i>String</i>	-- rules for each production
<b>prod</b> <i>Nil lhs.sh = Nil</i>	-- rule 1: def. of syn attr of LHS
<b>prod</b> <i>Cons tl.down = loc.hd</i>	-- rule 2: def. of inh <i>tl</i> with term <i>hd</i>
<i>lhs.sh = Cons lhs.down tl.sh</i>	-- rule 3: of syn attr of LHS

Figure 1.14: Example of an AG that shifts a character in a string.

**Notation for attribute grammars.** The above definitions introduce concepts that underly attribute grammar languages and implementations. Figure 1.13 gives a minimalistic language for the description of AGs, which is an extension of the language for context-free grammars in Figure 1.11. We explain some of its aspects below.

The notation<sup>5</sup> in Figure 1.13 consists of a collection of nonterminal declarations  $g$ , attribute declarations  $I$ , and semantics blocks  $S$ . Attributes are declared separately for each nonterminal and have a type associated with them. The right-hand side of a rule is an *expression*  $f [\bar{o}]$  in some formal language  $H$  with embedded references to attributes at identifier positions of  $H$  via attribute occurrences  $\bar{o}$ . The left-hand side of a rule is also an expression, but limited to patterns such as tuples. The use of expressions is slightly more flexible than just a function symbol.

For attribute occurrences  $t.x.y$ , we take the following notational conventions. The attribute kind  $t$  distinguishes inherited and synthesized attributes. This kind is always clear from the

<sup>5</sup> Note that we reused some letters here which we used before in a different context as the map  $I$  and the start symbols  $S$ .

## 1 Introduction

context thus we usually leave it unspecified in examples, unless we want to stress the difference. To refer to an inherited attribute  $y$  of a symbol named  $x$ , we use  $inh.x.y$  or simply  $x.y$ . To refer to a synthesized attribute  $y$  of a symbol named  $x$ , we use  $syn.x.y$  or simply  $x.y$ . To refer to a terminal named  $x$  we use the attribute occurrence  $loc.x.self$  or simply  $x.self$  (see also Section 1.3.6).

Informally, an AG in this notation is well-formed when the description can be translated to an AG, and that the types of various identifiers and expressions are correct.

Figure 1.14 shows an example where we define a transformation on strings where each character is shifted one position to the right. We use an inherited attribute *down* to represent the preceding character. Given a value 'd' as initial *down* value, the result for "Ag" is "dA". We show a more complex example in Section 1.3.11.

**Local attributes.** We also use the notation  $loc.loc.x$  to refer to a *local attribute* with the name  $x$ , which is an attribute defined by a rule of a production and is only in scope of that production. The name must be distinct from the name of a terminal symbol. Local attributes are typically used to represent common subexpressions.

**Syntactic sugar.** A terminal  $v$  can be encoded as a fresh nonterminal (Section 1.3.7) with a single synthesized attribute and a single  $\varepsilon$ -production that defines the attribute with the value  $\llbracket v \rrbracket$ . A local attribute can be encoded as a fresh nonterminal with a single inherited attribute and single synthesized attribute and a single  $\varepsilon$ -production that contains a rule that copies the value of the inherited attribute to the synthesized attribute. As convenient simplification, we therefore assume in the remainder of this chapter that terminal-leaves and local attributes are not present in AGs and ASTs, although we use terminals and local attributes in examples. In later chapters we take local attributes and terminals into account explicitly.

**Interfacing with the AG.** The evaluation of a tree described by the AG (which we discuss in Section 1.3.4) takes as input a record of values for the inherited attributes and results in a record with values of synthesized attributes.

### 1.3.2 Dependency Graphs

To describe how the values of attributes are actually computed, we consider the data dependencies induced by rules. The trees that we consider in this section are derivation trees generated by some AG  $(T, N, S, A, I, O, P, F)$ .

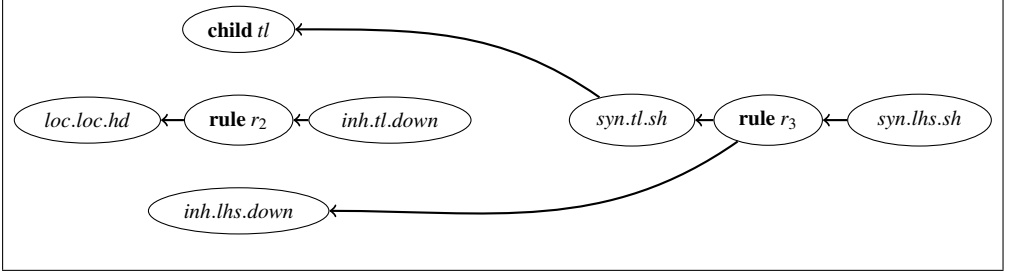
**Graphs of a production.** A rule  $\overline{o}_1 = f \overline{o}_2$  represents a data dependency of occurrences  $\overline{o}_1$  on occurrences  $\overline{o}_2$ , or equivalently, a flow of data from  $\overline{o}_2$  via  $f$  into  $\overline{o}_1$ . These dependencies form a graph.

**Definition** (Production dependency graph). A *Production Dependency Graph* (PDG) is a directed graph  $(V, E)$  associated with some production  $p$ . There is a one-on-one mapping between vertices  $d \in V$  (Figure 1.15), and the nonterminal children of  $p$ , the rules of  $p$  and attribute occurrences in rules of  $p$ . The edges  $E$  consists of:



$d ::= o$	-- attribute occurrence vertex
<b>rule</b> $x$	-- rule vertex with some distinct identifier $x$
<b>child</b> $x$	-- child vertex with $x$ the name of the child
$x$	-- identifiers

Figure 1.15: Syntax of vertices in a PDG.

Figure 1.16: Exemplary PDG of production *Cons*.

- For each vertex  $syn.x.y$  an edge to a vertex **child**  $x$ .
- For each rule  $[\overline{o}_1] = f[\overline{o}_2]$  (represented as vertex **rule**  $r$ ) an edge from **rule**  $r$  to vertex  $o$  for each  $o \in \overline{o}_2$ , and an edge from vertex  $o$  to **rule**  $r$  for each  $o \in \overline{o}_1$ .

Similarly, a *production data-flow graph* is a production dependency graph with the edges reversed.

Figure 1.16 shows the PDG of the production *Cons* of Figure 1.14. For simplicity, we modelled the terminal *hd* as a local attribute *loc.loc.hd*.

**Graph of a tree.** These graphs can be projected on each node of a tree and then combined to form a dependency graph for a tree, or a data-flow graph for a tree. The general idea is that we take the PDG of the root of the tree and then add the graph for each child of the root, which describe the dependencies of synthesized attributes of the child on inherited attributes of the child.

**Definition** (Tree dependency graph). For some annotated syntax tree  $t$  with associated production  $p$ , attribute association  $\alpha$ , and annotated subtrees  $t_1, \dots, t_k$  with corresponding nonterminal children  $c_1, \dots, c_k$  of  $p$ , the *tree dependency graph* is inductively defined as the union of the PDGs of  $t_1, \dots, t_k$  and the instantiation of the PDG of  $p$  by transforming (with preservation of edges) rule and child vertices to fresh vertices and each occurrence vertex  $o$  to vertex  $\alpha o$ . Similarly, a *tree data-flow graph* is a tree dependency graph with the edges reversed.

Given a tree, evaluation algorithms of AGs (Section 1.3.4) are traversals over the tree

## 1 Introduction

dependency graph expressible as tree traversals<sup>6</sup> over the tree that are described by a non-deterministic tree-walking automaton (Section 1.3.3). Traditionally, such traversals may be demand-driven (traversing the dependency graph based on which attributes are needed) or may be described by a deterministic tree-walking automaton.

We come back to evaluation algorithms in Section 1.3.4. We first consider static approximations of the dependency graphs which can be used to prove that none of the attributes have a cyclic definition, and are an essential ingredient for a description of the evaluation with a *deterministic* tree-walking automaton.

**Approximations.** Below, we consider abstract interpretations of AGs that construct dependency graphs that are a static approximation of the tree dependency graphs of collections of trees in certain contexts:

**Definition (Context).** A *context* is a symbol  $C$  of some fixed set of symbols  $\Omega$  given per application and grammar. A context represents an additional set of invariants imposed on a tree.

Concretely, the invariants represented by a context may include that:

- The tree is associated with a certain nonterminal or production;
- The tree occurs as a subtree at certain position of a parent;
- The tree has attributes that are used according to some protocol [Farrow, 1984].

A collection of trees in a context share a common structure. By distinguishing contexts, we may consider projections of the tree dependency graphs of a collection of trees on the common attributes (of the root) that each tree in the collection has, so that the edges are superset of the projected edges of each individual tree. The projection-operation distributes over graph union, which ensures that we can work with approximations of projections of graphs of subtrees.

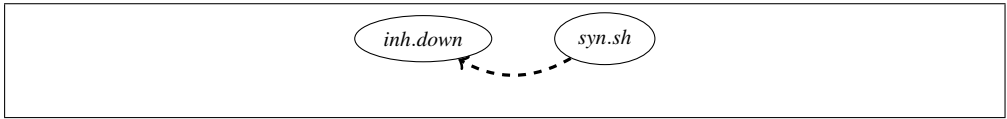
**Definition (I/O graph).** An *I/O graph* of some nonterminal  $n \in N$  is directed graph where the vertices consist of the attributes  $a \in I_n \cup O_n$  and the edges represent either (indirect) dependencies or data flow between attributes of some trees that have  $n$  as root. An *I/O dependency graph* is an *I/O graph* where the edges represent data dependencies, and in an *I/O flow graph* the edges represent data flow.

Note that the above definition does not specify which edges are included in an I/O graph, but only specifies what the edges represent. We later give a consistency condition that specifies which edges must minimally be present.

**Definition (Nonterminal dependency graph).** The *Nonterminal Dependency Graph* (NDG) of nonterminal  $n$  is the I/O dependency graph of  $n$  that approximates the dependencies between attributes of any *tree* associated with  $n$ .

---

<sup>6</sup> The shape of a tree dependency graph ensures that a description with a tree traversal is possible. The dependencies are properly nested: on a path from a synthesized attribute of a child to some (indirect) dependency, an inherited attribute of the child occurs before any attribute of a sibling or parent.



**Figure 1.17:** An I/O graph of nonterminal *String*.

An I/O graph of some nonterminal  $n$  serves as an approximation of the dependencies between attributes of *some* trees (context dependent) that are associated with  $n$ . In contrast, an NDG is a single, context-independent approximation of dependencies between attributes of some nonterminal.

**Definition** (Augmented PDG). The *augmented PDG* of a PDG  $G$  of some production  $p$  and I/O graphs of the nonterminal children of the production, is  $G$  with edges added between attributes of children so that if there exists an edge between attributes of the given I/O graph of some child, then this edge also exists between the attributes of the child in the augmented PDG, and vice versa<sup>7</sup>.

An augmented PDG of some production  $p$  serves as an approximation of the dependences between attributes of *some* trees that are associated with  $p$ . It is a PDG parameterized with the dependencies induced by the subtrees.

**Collections of graphs.** To specify which edges are part of the I/O graphs and augmented PDGs we consider some properties of collections of these graphs.

**Definition** (Consistent approximations). A collection of I/O graphs and augmented PDGs is *consistent* when each I/O graph of some nonterminal  $n$  in some context  $\mathcal{C}$  is a projection of the union of the augmented PDGs in context  $\mathcal{C}$  of the productions associated with  $n$ .

**Definition** (Complete approximations). A collection of I/O graphs and augmented PDGs is *complete* when for each production  $p$  the collection includes an augmented PDG and corresponding I/O graphs for each combination of contexts that the children of  $p$  can occur in.

**Definition** (Smaller approximations). Graph  $A$  is *smaller* than graph  $B$  if  $A^*$  is a subgraph of  $B^*$ , where  $A^*$  and  $B^*$  are the respective transitively closed graphs of  $A$  and  $B$ .

Figure 1.17 gives an example of an I/O graph that is part of some collection that satisfies the above properties based on the AG in Figure 1.14. It is an I/O graph of the nonterminal *String* in the context of being the child *tl* of production *Cons*. The dependency from *syn.sh* on *inh.down* is induced by the production *Cons*. If Figure 1.16 would include an edge from *syn.tl.sh* to *inh.tl.down*, it would be an augmented PDG parameterized with the I/O graph.

<sup>7</sup>We specify here that the dependencies between attributes of a child in an augmented PDG match exactly to the dependencies between the attributes of the corresponding I/O graph that the augmented PDG is parameterized with, thus that edges of the PDG may impose constraints on the I/O graphs of the children.

## 1 Introduction

**Cycle analysis.** Cycle analysis of AGs is an abstract interpretation [Nielson et al., 1999] that approximates the tree dependency graph of any tree by constructing a complete and consistent collection of I/O graphs and augmented PDGs. The consistency and completeness requirements lead to a set of mutually recursive equations for which we want to compute a least solution. Such a solution is obtained with fixpoint iteration. In Chapter 4 we consider a concrete cycle analysis; here we look only at the general structure of such analyses.

Various approaches distinguish different contexts which influence the accuracy and complexity of the approximations:

**Uniform AG.** Knuth [1968] distinguishes as contexts the nonterminals to which trees are associated, and the different positions in the right-hand sides of productions where nonterminals can occur in. This approach<sup>8</sup> leads to one I/O graph per nonterminal and one augmented PDG per production. The AG is *uniform* or *absolutely non-circular* if the augmented PDGs in this flavor are acyclic.

**Well-defined AG.** Knuth [1971] additionally distinguishes the production to which the tree is associated. This approach leads to one I/O graph per nonterminal and production, and an exponential number of augmented PDG, one for each combination of children with productions. An AG is *well-defined* if for any tree the tree dependency graph is cycle-free, which is the case when the augmented PDGs in this flavor are acyclic.

**Ordered AG.** Kastens [1980] distinguishes as contexts only the nonterminals to which trees are associated, but does not differentiate the occurrences of nonterminals in the right-hand sides of productions.

In our experience with UUAG and in agreement with observations by Rähkä and Saarinen [1982], an AG is in practice also ordered when it is well-defined.

### 1.3.3 Tree-Walking Automata

Tree-walking automata arose from tree language theory and were introduced by Aho and Ullman [1969]. A tree-walking automaton (TWA) is device that walks over a tree in a contiguous manner and is accompanied by a state machine that describes how the nodes of the tree change their state upon each visit and whether the device goes up to the parent or goes down to one of the children as the next step. Section 1.3.4 uses TWAs to describe evaluation algorithms of AGs. We keep here a simplistic presentation; there exist many extensions that increase the expressiveness of these automata, such as pebbles [Engelfriet and Hoogeboom, 1999].

**Definition** (Tree-walking automaton). A TWA for some AST is a tuple  $(V, Q, I, F, \delta)$ , with an alphabet  $V$  of node labels, a finite set of states  $Q$ , an initial state  $I \in Q$ , a set of final states  $F$ , and a transition relation  $\delta \subseteq (V \times Q \times Q \times C \times Q)$ , where  $C = \{up, down_0, \dots, down_k\}$  is a set of commands and  $k$  is the maximum branching factor of nodes in the AST. There exists a

---

<sup>8</sup> The actual approach does not distinguish different positions but instead instantiates augmented PDGs with I/O graphs (thus copies them).

$p ::= \mathbf{plans} P : N \bar{v}$	-- execution plan for production $P$ of nonterminal $N$
$v ::= \mathbf{visit} C \bar{i} \bar{s} \bar{b}$	-- a visit in context $C$ that may need $\bar{i}$ and can produce $\bar{s}$
$b ::= r$	-- evaluation rule
$\mathbf{invoke} x C \bar{i} \bar{s}$	-- a <i>down</i> $x$ in context $C$ , which can provide $\bar{i}$ to $x$ and expects $\bar{s}$
$x$	-- child name
$C$	-- context identifier
$i, s, r$	-- as defined in Figure 1.13

**Figure 1.18:** The execution plan language.

one-to-one relation between productions and symbols in  $V$  so that each node is labelled with a  $v \in V$  depending on the production associated with the node.

A tuple  $(v, q, q_0, c, q') \in \delta$  represents a transition from a node labelled  $v$  in state  $q$  to a state  $q'$ , and moving to the node according to  $c$ . The automaton keeps track of a bit of history:  $q_0$  is the state of the node that caused the transition to the current node.

**Definition** (Deterministic tree-walking automaton). A *deterministic TWA* is a TWA where the transition relation is a function  $\delta :: V \rightarrow Q \rightarrow Q \rightarrow (C, Q)$ . Otherwise the TWA is *non-deterministic*.

**Acceptance.** Initially each node in the AST is associated with the initial state  $I$ . The automaton starts at the root of the tree and stops if no step can be taken anymore. The tree is accepted if the automaton ends with the root having an associated state in  $F$ . With each step, the automaton visits a node. If the automaton is at a node with label  $v$  and associated state  $q$ , and previously visited a node in state  $q_0$ , then the automaton chooses a step  $c$  and new state  $q'$  so that  $(v, q, q_0, c, q') \in \delta$ , or the automaton stops if no such step exists. In the former case, the automaton updates the state of the node to  $q'$  and visits the parent if  $c = up$  or visits child  $i$  if  $c = down_i$ .

**Evaluation of rules.** An actual AG evaluation algorithm does not only traverse the tree but also needs to apply rules to compute attributes. Thus, in an actual implementation, the automaton also applies a subset  $\gamma(v, q)$  of the production's rules upon making a transition to state  $q$  at a node with label  $v$ .

**Definition.** Visit The TWA *visits* a node  $n$  if it arrives at  $n$  and executes  $\gamma(v, q)$  where  $v$  is the label of  $n$  and  $q$  is the state of  $n$ .

**Implementation with the Zipper.** Various forms of deterministic TWAs can be implemented in a purely functional programming language using the *zippers* [Huet, 1997]. Such an approach models the imperative updates of the automaton to the state.

```

plans Nil : String
  visit AnyCtx inh down :: Char syn sh :: String
    lhs.sh = Nil

plans Cons : String
  visit AnyCtx inh down :: Char syn sh :: String
    tl.down = loc.hd
  invoke tl AnyCtx inh down :: Char syn sh :: String
    lhs.sh = Cons lhs.down tl.sh

```

**Figure 1.19:** Exemplary execution plans of nonterminal *String*.

**Execution plans.** Figure 1.18 introduces a language of execution plans  $\bar{p}$  for the description of the transition relation of TWAs, of which we explain some aspects below. Figure 1.19 shows an example. The language is not expressive enough to describe all transition relations, but it suffices for a description of an AGs evaluation.

A collection of plans-blocks represents the transition *relation*  $\delta$ . Since a context is an agreement between parent and child, a context models the  $q_0$  parameter of  $\delta$ . A plans-block is associated to a unique production  $P$  and consists of a number of visit-blocks. A visit-block  $v$  describes visits to the node in context  $C$ , and thus represents a subset  $\delta_{P_C}^v$  of  $\delta_{P_C}$ . Let  $\bar{r}$  be the rules that can be evaluated as a consequence of the tree walk taking transitions from  $\delta_P^v$ . Then the inherited attributes  $\bar{i}$  of the current node may be needed in the evaluation of  $\bar{r}$  and the synthesized attributes  $\bar{s}$  can be computed by  $\bar{r}$ . An invoke-rule represents possible transitions to some child  $x$  of  $P$  in some context  $C$  so that values of inherited attributes  $\bar{i}$  of  $x$  can be provided, and values of synthesized attributes  $\bar{s}$  of  $x$  may be needed by the current node.

Given an acyclic PDG of  $P$ , the relation  $\delta_P$  and the accompanying subset of  $\gamma$  can be generated. This procedure is left as an exercise to the reader; we note that the number of states is possibly exponential in the size of the productions and their children, and the order of appearance of rules is irrelevant for the translation.

A collection of plain-blocks may represent also a transition *function*  $\delta$ . In this case, each visit-block  $v$  represents a distinct  $\delta_{P_C}^v \subseteq \delta_{P_C}$  with precisely  $1 + n$  elements (of which element  $i + 1$  can be thought of as the continuation after visiting child number  $i$ ), where  $n$  is the number of invoke-rules in a visit-block. The rules must occur in define-before-use order. The example in Figure 1.19 satisfies these constraints.

**Implementations of execution plans.** In comparison to the Zipper, there are less ‘imperative’ and more efficient encodings of TWAs in functional languages. If we consider TWAs with transition relations that do not make use of the  $q_0$  parameter, and thus do not distinguish contexts, Swierstra and Alcocer [1998] presented an approach by exploiting laziness (for nondeterministic TWAs) which we sketch in Section 1.3.4, and Saraiva and Swierstra [1999] presented visit functions, which are coroutines encoded as continuations (for deterministic

TWAs with a total order imposed on visits which we sketch in Section 1.3.5).

**Implementations described in this thesis.** In Chapter 3 we build upon visit functions. In Chapter 4 we show that we can represent transition relations that use contexts. In later chapters we look at extensions to AGs that have accompanying evaluation algorithms that cannot be described with a deterministic TWA. For example, in Chapter 5 we allow tree walks directed by values of attributes, and in Chapter 6 we allow tree walks to jump back and forth saved positions.

### 1.3.4 Demand-driven Attribute Evaluation

We can express demand-driven attribute evaluation by mapping an AG onto an execution plan that describes a nondeterministic TWA (Section 1.3.3). This translation is straightforward:

- From each production an execution plan is derived with a single visit that lists all attributes of the productions left-hand side.
- The visit-block contains the rules of the production, and an invoke-rule per child which lists all the attributes of the child.

The attributes are then computed by running the described TWA.

An execution plan in the above form has a straightforward translation to algebras Haskell. We sketch this translation. It is optional background material, but is not required for the understanding of later chapters.

**Catamorphisms.** Let  $F$  be an endofunctor so that data constructors describing the AST  $A$  form the initial  $F$ -algebra. The execution plans can be mapped straightforwardly to an  $F$ -algebra  $\phi$  so that  $\text{cata } \phi A$  is a function  $g$  (which we call the *semantic result* or *semantic tree*) that takes an argument for each inherited attribute of the root of  $A$ , and provides a result for each synthesized attribute of the root. Function  $g$  encodes the tree dependency graph of  $\text{ast}$  (Section 1.3.2), and lazy evaluation acts as nondeterministic TWA, with the additional feature that each rule in the execution plan is at most executed once. Effectively, the functional program is a term-graph representation of the dependency graph, and evaluation, rewriting this term-graph [van Eekelen et al., 1996], results in the values of synthesized attributes.

**Haskell translation.** Swierstra and Alcocer [1998] showed how to express the algebra in Haskell as a function (called a *semantic function*<sup>9</sup>). We demonstrate this approach based on the example in Figure 1.14.

Figure 1.20 gives a sketch of the translation to Haskell code. We explain some aspects of this example below.

For each nonterminal  $N$  a type  $T\_ \langle N \rangle$  for the semantic result of a tree associated with  $N$  is generated, which is  $T\_String$  in the example. Nonterminal  $String$  has an inherited attribute of

<sup>9</sup> Depending on the context, a semantic function may be the function in a productions rule or a function corresponding to some data structure in an algebra.

```

type T_String = Char → String           -- type of the node's semantics
cata_String :: String → T_String         -- maps AST to semantic AST
cata_String Nil           = sem_Nil
cata_String (Cons hd tl) = sem_Cons hd (cata_String tl)
sem_Nil :: T_String                       -- production without children
sem_Nil lhs_down = (lhs_sh) where ...    -- note that the T-type is a function
sem_Cons :: Char → T_String → T_String -- production with two children
sem_Cons loc_hd loc_tl lhs_down = (lhs_sh) where ...

```

**Figure 1.20:** Sketch of the algebra.

```

sem_Cons loc_hd loc_tl = λlhs_down →
  let tl_down = loc_hd           -- transcription of the first rule
      (tl_sh) = loc_tl tl_down    -- recursive call to child tl
      lhs_sh = Cons lhs_down tl_sh -- transcription of the second rule
  in (lhs_sh)

```

**Figure 1.21:** The body of *sem\_Cons*.

type *Char* and a synthesized attribute of type *String*, hence the type of *T\_String* is *Char* → *String*.

The *cata*-function associates a semantics with each constructor of the AST. It replaces a constructor *P* with its semantic variant *sem\_⟨P⟩*.

**Semantic functions.** Figure 1.21 sketches the generated semantic function *sem\_Cons*. The body of a semantic function encodes the productions rules and the invoke-rules as given in the execution plan. The encoding of the rules is straightforward. An invoke-rule with some child *tl* is translated to a recursive call to the parameter that represents the semantic tree of *tl*. The call is parameterized with values of the inherited attributes of *tl*, and a pattern match against the results extracts the values of the synthesized attribute of *tl*.

**Remarks.** Demand-driven evaluation of AGs is popular in current AG systems. An advantage of on-demand evaluation is that it does not require an abstract interpretation as part of its implementation, and works in combination with extensions such as remote reference attributes [Magnusson and Hedin, 2007].

Another advantage is that attributes are not computed when their values are not needed at runtime. Demand-driven evaluation may produce values of attributes even in the presence of cyclic attribute dependencies.



A disadvantage of demand-driven evaluation is the potential high space requirements also known as space leaks.

The approach based on lazy evaluation goes further than demand-driven attribute evaluation because the set of required attributes depends on how the values of attributes are inspected. In fact, using lazy lists it is possible to associate countable infinite attributes with a nonterminal.

### 1.3.5 Statically Ordered Attribute Evaluation

We can express a statically ordered evaluation of AGs by mapping the AG onto a deterministic TWA (Section 1.3.3). Depending on what contexts are distinguished and the strictness properties of the rules, there may be values computed for attributes that are not needed for the result, or which are only needed later. Therefore, we will call this also eager, greedy or strict evaluation of AGs.

**Definition** (Multi-visit AG). A *multi-visit AG* is an AG for which a statically ordered evaluation strategy is possible.

Multi-visit AGs play an important role in this thesis because they make the notion of phasing explicit, which is useful for reasoning about what parts of the tree have been investigated and constructed so far.

**Attribute scheduling.** To map an AG to an execution plan of a deterministic TWA, we need to determine for each production  $P$  how trees are visited that are associated to  $P$ . For each visit, we need to determine which rules to apply and how the children of  $P$  are visited. This process requires acyclic augmented PDGs.

**Scheduling algorithms.** Kastens [1980] presented an approach that attempts to derive a smallest single sequence  $\Delta_N$  of visits per nonterminal  $N$  so that the attributes of a child with some nonterminal  $N$  can be computed by visiting the child according to some prefix of  $\Delta_N$ . Effectively, this approach does not distinguish any contexts. The approach entails adding edges to the I/O graphs of  $N$  so they are equal and form a total order on the attributes, which is possible for most AGs in practice, but the approach of Kastens sometimes needs help in the form of additional attribute dependencies to accomplish this.

Kennedy and Warren [1976] presented an approach that works for any absolutely non-circular AG. Their approach distinguishes protocols as contexts, which are the possible orders in which the parent provides inherited attributes and demands synthesized attributes. In Chapter 4 we investigate this approach in-depth and present a translation to Haskell.

**Coroutines.** A deterministic TWA can be implemented with coroutines [Warren, 1976].

**Definition** (Coroutine). A *coroutine* [Marlin, 1980] is a function that can pause during its execution and return results to the caller. It may be parameterized with additional arguments when resumed by the caller. A *generator* is a coroutine that does not take additional arguments.

## 1 Introduction

When no contexts are distinguished, such coroutines can be encoded in a purely functional language as *visit functions* [Saraiva and Swierstra, 1999]. The body of such a function builds a continuation that is used for a subsequent call and returns it as part of the result. From the caller's perspective:

```
...
f_visit1 = s.f -- assuming s.f is the coroutine of a child f
(f_syn1, f_syn2, f_visit2) = f_visit1 f_inh1 f_inh2
(f_syn3, f_syn4, f_visit3) = f_visit2 f_inh2 f_inh2
...
```

The callee has the following structure where the dots represent the usual encoding of the rules that are scheduled to a particular visit:

```
sem_(P) _I s_child1 s_child2 = lhs_visit1 where
  lhs_visit1 inh1 ... inh2 = (syn1, ..., syn2, lhs_visit2) where
  ...
  lhs_visit2 inh3 ... inh4 = (syn3, ..., syn4, lhs_visit3) where
  ...
```

We explain this encoding in great detail throughout this thesis. Many chapters of this thesis present variants of this encoding.

**Remarks.** Historically, statically ordered attribute evaluation results in faster code and less memory usage. Also, recent developments on multi-core computing may give renewed interest in visit sequences with respect to parallel evaluation [Wang and Ye, 1991].

### 1.3.6 Incremental Descriptions

Our formalism allows us to write various declarations that together form an AG in any order. This is a consequence of the purely functional relation between attributes. It is an important property of AGs, because it allows us to incrementally and separately describe AGs. In this thesis, we make repeatedly use of this feature to eliminate common patterns from examples. The separate descriptions are simply merged by string concatenation.

**Incremental notation.** An AG description is incremental: nonterminals, productions, children, attributes and rules may be declared separately. At the same time, we may declare productions and attributes for multiple nonterminals, and children and rules for multiple productions.

Figure 1.22 gives a number of examples. A declaration of a nonterminal and production may appear multiple times and provide additional declarations. Nonterminal *Expr* has only one production *App* but its contents are determined by several declarations.

The type **self** is special and represents the type of the actual nonterminal of the description the **self** appears in. The function *mkApp* must thus be an overloaded function that works both on *Exprs* and on *Types*.

```

grammar Expr                                -- productions for Expr
  prod App   term impred :: Bool           -- with terminal for App
  prod Var   term nm    :: Name           -- with terminal for App
  prod App Var term uid  :: Int           -- extra terminal for App and Var

grammar Expr Type                             -- productions for multiple nonterminals
  prod App   nonterm f, a : self         -- multiple nonterminals of the same type
attr Expr   inh x, y :: Int             -- multiple attributes of the same type
attr Expr Type syn output :: self       -- attributes for multiple nonterminals
sem Expr Type                                 -- rules for multiple nonterminals
  prod App   lhs.output = f.output 'mkApp' a.output

sem Expr                                       -- rule for multiple productions
  prod App Var lhs.x = 1

```

**Figure 1.22:** Examples of incremental notation.

The merging process is straightforward<sup>10</sup>. A duplicate declaration of an attribute of the same nonterminal is not allowed and considered a static error. Similarly, after merging, attributes may not be defined by more than one rule, and each child of a production must be defined once.

**Nonterminal sets.** To aid the definition of attributes on many nonterminals, we may use nonterminal sets. We use nonterminal sets often in actual code but only sporadically in this thesis.

**Definition** (Nonterminal set). A *nonterminal set* is a nonterminal name that represents one or more other nonterminals or sets.

For example, we define a nonterminal name *AllExpr*, which actually stands for *Expr* and *Decl*. When we declare attributes on *AllExpr*, these are actually declared for *Expr* and *Decl*:

```

set AllExpr : Expr Decl  -- AllExpr includes  $Expr \cup Decl$ .

```

Nonterminal sets are extensible: a set declaration of some set  $N$  may appear multiple times in an AG description. Additionally, notation for set union and set difference may be used to define sets. Determining sets is a straightforward fixpoint computation.

**Nonterminal inheritance.** A nonterminal may masquerade as a set. If due to a set declaration, such a set includes other sets and nonterminals, the nonterminal inherits their attributes, productions, and rules.

<sup>10</sup> The interested reader may take a look at `Transform.ag` in the `uuagc` project for a merge algorithm. This algorithm also allows some declarations to overwrite previous declarations.

### 1.3.7 Higher-Order Children and Attributes

**Definition** (Semantics of a child). The *semantics of a child* with a nonterminal  $N$  is a semantic tree associated to nonterminal  $N$ . The decorations still have to be given.

**Definition** (Higher-order child). A *higher-order child* is a child with a semantics determined by the value of an attribute.

A conventional child is determined by syntax, whereas a higher-order child is determined by an attribute. Higher-order children are also known as higher-order attributes or nonterminal attributes. The notion ‘higher order’ originates from being able to pass the semantics of children around as first class values.

*Higher-Order* AGs (HOAGs) [Vogt et al., 1989] support higher-order children. As part of this thesis, we implemented this feature in UUAG [Löh et al., 1998]<sup>11</sup>. Higher-order children play an essential role in this thesis: with such children we can dynamically grow the tree (e.g. a proof tree) instead of being limited to a fixed tree (e.g. the parse tree).

**Children defined by rules.** In a conventional AG, the semantics of a child of a production is determined prior to attribute evaluation. In a Higher-Order AG (HOAG), additional higher-order children may be declared for a production. Their semantics is the value of an attribute, or alternatively, the outcome of evaluating a rule:

**child**  $x:N = f [\bar{a}]$  -- rule that introduces a child  $x$

The expression  $f [\bar{a}]$  evaluates to the semantics for  $x$  as the following example demonstrates:

**child**  $x:String = sem\_Nil$  -- declares a child  $x$  that is defined by  $sem\_Nil$   
 $x.down = 'z'$  -- inherited attr of  $x$

**Implementation.** We see in later chapters how child-rules can be implemented. In the translation to Haskell as sketched in Section 1.3.4, the semantics of a child is a function from inherited to synthesized attributes, and each child is translated to a function call. A child-rule in this section is a conventional evaluation rule that defines some local attribute, where the local attribute is used as the function to call:

$loc\_x = sem\_Nil$  -- local attr determines the semantics of  $x$   
 $x.down = 'z'$  -- defines inherited attr of  $x$   
 $x.sh = loc\_x x.down$  -- call to child  $x$

<sup>11</sup> The syntax that we use here deviates slightly from the actual syntax of higher-order children in UUAG. A type signature **inst**. $x:N$  declares a child  $x$ , and a conventional rule must define the attribute **inst**. $x$  with the semantics for  $x$ . In addition, when  $x$  already exists, its definition is a function that transforms the original semantics of  $x$ .

**Desugaring.** HOAGs can be used to desugar an AST. As example of an HOAG, suppose that *String* has a special production *Single* for single-character strings. Instead of defining the semantics directly for *Single*, we add a child *repl* that represents the string in terms of *Cons* and *Nil*:

```

grammar String prod Single   term x::Char   -- the Single production
sem String prod Single         -- and its semantics
  child repl:String = sem_Cons loc.x sem_Nil -- higher-order child
  repl.down           = lhs.down             -- inh attr of child repl
  lhs.sh              = repl.sh             -- syn attr of child repl

```

**Multi-visit AGs as HOAGs.** In a multi-visit AG, a child may be visited multiple times to compute some of the attributes. Such an AG can be encoded as a HOAG, which we show below. As we see in later chapters, we worked on a core language that can represent such AGs, and the question arose whether to use HOAGs as a target language. We did not do this because of other requirements, but we present the translation anyway since it may give some insight in how we organized the visit functions earlier.

We may encode multiple visits to some child *c* as a single visit to child *c* that only requires the inherited attributes of the first visit and only provides the synthesized attributes of the first visit. Additionally, it produces an attribute *c.cont* that represents the semantics of *c* after the visit. For the second visit, we use a higher-order child *c<sub>2</sub>* with *c.cont* as semantics. We then visit *c<sub>2</sub>* to provide/obtain the attributes of the second visit. For the next visit we use *c<sub>2</sub>.cont*, etc. This approach requires the introduction of a potential large number of new nonterminals.

For a nonterminal *N* with *m* visits, we introduce the nonterminals *N<sub>1</sub>, ..., N<sub>m</sub>*, such that each nonterminal *N<sub>i</sub>* has the inherited and synthesized attributes as associated to visit *i* of nonterminal *N*. In addition, *N<sub>i</sub>* (for *i < m*), has an extra synthesized *continuation* attribute *cont* that contains the semantics of *N<sub>i+1</sub>*.

For a production *P* we introduce the productions *P<sub>1</sub>, ..., P<sub>m</sub>*. The terminals of production *P<sub>i</sub>* represent the decorations as available prior to visit *i*. Thus, *P<sub>1</sub>* consists of the original terminals and nonterminals of *P*, and *P<sub>i</sub>* (for *i > 1*) consists of the terminals of *P<sub>i-1</sub>* and additionally has terminals which encode the attributes computed in visit *i*. As optimization the terminals that are not needed in later visits can be omitted from *P<sub>i</sub>*.

The semantics for *P<sub>i</sub>* consists of the rules in *P*'s plan for visit *i* (modulo renaming of attribute references). Additionally, a rule is added which computes the semantics of *P<sub>i+1</sub>* and stores it in attribute *lhs.cont*.

In this translation, a visit to a child (for *i > 1*) is thus represented as a higher-order child that is instantiated by the continuation attribute produced by the previous visit.

### 1.3.8 Circular Reference Attributes

Reference Attributed AGs (RAGs) are an extension of attribute grammars with (remote) reference attributes [Magnusson and Hedin, 2007]. This is a common extension of AGs that utilize a demand-driven evaluation algorithm. The extension allows subtrees to be passed

## 1 Introduction

around in attributes. Normally, the attributes of such a subtree  $T$  can be used and defined only by the direct parent of  $T$ . However, with the extension, the attributes of  $T$  may be used and defined by any rule holding a reference to  $T$ . Such an attribute is said to be accessed *remotely*.

**Graph structure.** With this extension, the nodes are organized in a graph structure instead of a tree. Calculations over graphs often require some form of repetition, which can be encoded with cyclic attributes. Values of cyclic attributes can be computed if the demand-driven evaluation is extended with fixpoint iteration, and the attributes are given an initial value. The computation terminates if the rules are monotonic and each ascending chain of attribute values stabilizes.

**Advantages and disadvantages.** Reference attributes provide a convenient way of transporting information from one location in a tree to another location. Also, the extension allows more analyses to be modelled with AGs, such as abstract interpretations, which are typically fixpoint iterations over graphs. This expressive power comes with a price: the well-definedness of an AG cannot be statically verified in general (Section 1.3.2). Moreover, if values of *inherited* attributes can be defined remotely, well-formedness of the AG cannot be checked statically, which has as consequence that a straightforward mapping (such as in Section 1.3.4) to a purely functional language is not possible<sup>12</sup>.

### 1.3.9 Correspondences between AGs, HOAGs, Monads, and Arrows

In later chapters, we translate AGs to a monadic target language [Meijer and Jeuring, 1995] and also consider AGs translated to Arrows [Hughes, 2004]. Being able to structure a computation as a monad or arrow allows reflection on the structure of the computation. We use such introspection in Chapter 6 to implement a step-wise evaluation strategy.

For example, consider the code of production *Cons* of the example in Section 1.3.1. In its present formulation, it can be evaluated in a strict fashion:

```
sem.Cons loc_hd loc_tl lhs_down = (lhs_sh) where
  tl_down = loc_hd
  (tl_sh)  = loc_tl tl_down
  lhs_sh   = Cons lhs_down tl_sh
```

We rewrite this code using arrow notation [Paterson, 2001] and call the result an execution plan:

```
sem.Cons field_hd loc_tl = proc lhs_down → do
  tl_down ← fcopy < field_hd           -- transcription of the first rule
  tl_sh   ← loc_tl < tl_down           -- invoke arrow of child tl
  lhs_sh  ← fcons < (lhs_down, tl_sh) -- transcription of the second rule
  returnA < lhs_sh                     -- output
```

---

<sup>12</sup> We implemented synthesized reference attributes in UUAG for lazily evaluated grammars.

where  $f_{copy} = id$   
 $f_{cons} = uncurry\ Cons$

The translation into arrow notation essentially desugars the above code into point-free style, thus in a linear composition of the rule functions that is interspersed with combinators to rearrange the intermediate attribute values.

Can we represent any AG in this way? When we limit the expressiveness of the language of the rules to tree constructions only, conventional AGs can express primitive recursive functions whereas HOAGs can express all computable functions. This difference can be observed when translating to arrows. In this setting, the semantics of a tree is an arrow that takes a tuple of inherited attributes as input and produces a tuple of synthesized attributes as output. A conventional AG is expressible as a plain arrow, but a HOAG requires the generalization to a monad, and AGs that are not statically ordered may require a feedback-loop.

Introspection of the arrow is possible using defunctionalization [Reynolds, 1998]. Such an arrow actually encodes the tree dependency graph (Section 1.3.2), thus introspection on this structure allows a runtime optimization (e.g. elimination of identity functions) of this graph which may be useful when traversing parts of the graph several times (Chapter 5).

### 1.3.10 Specification of Typing Relations

In the preceding sections, we discussed several AG features. We will now make the connection with type systems. A reason why attribute grammars are a convenient formalism to describe type systems is that we can see a specification with type rules as an attribute grammar where the typing relations are attributed nonterminals, the type rules form the productions, and the judgments form the symbols with equations between the attributes. The AG thus describes the structure of derivation trees.

To map an AG to type rules, we need to translate nonterminals, productions and rules. A nonterminal  $N$  with inherited attributes  $I_N$  and synthesized attributes  $O_n$  translates to a typing relation of the form  $I_N \vdash_N O_n$ . Note that the distinction between inherited and synthesized gets lost in the translation, although we use the notational convention that supposed inputs are to the left of the turn style and supposed outputs are to the right. Without going in details, a production  $p = u \rightarrow \bar{w} \cdot \bar{r} \cdot X$  translates to a type rule with a judgments for each symbol in  $\bar{w}$ , equality judgments for each rule in  $\bar{r}$ , and a conclusion for  $u$ . The actual translation is quite similar to the translation to a lazy functional program, which we already showed in Section 1.3.4.

If we take the inverse of the above translation, we arrive at a translation from type rules to AGs. This process, however, is non-trivial, which is not surprising because there exists no general inference algorithm. Thus, to do so, we need to identify which attributes are inherited and which are synthesized, and with what computable function to represent equality judgments. However, as we come back to in Section 1.6, we can use such an improper AG as a starting point for an inference algorithm, and add information to make it a proper AG. Such an AG then has the desirable property that its basis directly corresponds to the specification.

In the next sections, we give an AG implementation of a DHM type inferencer, and show some attribute grammar features that are useful when using AGs to describe derivation trees.

```

grammar Expr                                -- abstract grammar for expressions
  prod Var term   x :: Name                 -- the identifier x
  prod App nonterm f : Expr                 -- left-hand side of application
                        nonterm a : Expr     -- right-hand side of application
  prod Lam term   x :: Name                 -- the identifier x bound by the lambda
                        nonterm b : Expr     -- the body of the lambda
  prod Let term   x : Expr                 -- the name of the binding
                        nonterm e, b : Expr    -- the binding and body

attr Expr
  inh env :: Env                          -- inherited environment
  chn subst :: Subst                       -- chained attr: both inh and syn
  syn ty  :: Ty                            -- synthesized type
  syn errs :: Errs                        -- collection of type errors

```

Figure 1.23: DHM grammar and attributes.

### 1.3.11 Damas-Hindley-Milner Inference

As an example of a type inference algorithm written with attribute grammars, we give an AG implementation of the DHM algorithm (Section 1.2.6) by using the same approach as presented by Dijkstra and Swierstra [2004].

**Definition** (Chained attribute). A *threaded attribute* or *chained attribute* stands for both an inherited and a synthesized attribute with the same name.

The environment is modeled as an inherited attribute, errors as a synthesized attribute, and the substitution as a chained attribute.

For the type we have two options. Either the parent passes an expected type to the child that is further constrained by the child, or the child passes up an inferred type that is further constrained by the parent. When type annotations are part of the language, the former approach detects type errors faster. The information from these type annotations can then be given to a child, instead of verifying the resulting type after the fact. However, for this example it does not matter, thus we take the latter approach, which we also used for the monadic implementation of DHM in Section 1.2.6.

Figure 1.23 shows the grammar and attributes of the example. The chained attribute *subst* is shorthand for an inherited and synthesized with both *subst* as name. To make it clear which of the two attributes we intend, we explicitly prepend *syn* and *inh* to the name in the rules.

The rules in Figure 1.24 describe how the environment is passed top-down through the tree, how the substitution is threaded in-order through the tree, and how errors are collected bottom-up. Since functions such as *instantiate* and *unify* produce an updated substitution, it is the threading of the substitution that determines in what order the effects of these operations are visible in the substitution. The substitution needs to be threaded carefully in order not to lose any constraints on type variables.



<b>sem</b> <i>Expr</i>		
<b>prod</b> <i>Var</i>	<i>loc.scheme</i>	= <i>lookup loc.x lhs.env</i>
	( <i>lhs.ty, syn.lhs.subst</i> )	= <i>instantiate loc.scheme inh.lhs.subst</i>
	<i>lhs.errs</i>	= $\emptyset$
<b>prod</b> <i>App</i>	<i>a.env</i>	= <i>lhs.env</i>
	<i>f.env</i>	= <i>lhs.env</i>
	( <i>loc.res, inh.a.subst</i> )	= <i>fresh inh.lhs.subst</i>
	<i>inh.f.subst</i>	= <i>syn.a.subst</i>
	( <i>loc.errs, syn.lhs.subst</i> )	= <i>unify f.ty (a.ty <math>\rightarrow</math> loc.res) syn.f.subst</i>
	<i>lhs.errs</i>	= <i>f.errs ++ a.errs ++ loc.errs</i>
<b>prod</b> <i>Lam</i>	<i>b.env</i>	= <i>insert loc.x loc.argty lhs.env</i>
	( <i>loc.argty, inh.b.subst</i> )	= <i>fresh inh.lhs.subst</i>
	<i>lhs.ty</i>	= <i>loc.argty <math>\rightarrow</math> b.ty</i>
	<i>syn.lhs.subst</i>	= <i>syn.b.subst</i>
	<i>lhs.errs</i>	= <i>b.errs</i>
<b>prod</b> <i>Let</i>	<i>e.env</i>	= <i>lhs.env</i>
	<i>b.env</i>	= <i>insert loc.x loc.scheme lhs.env</i>
	<i>loc.scheme</i>	= <i>generalize lhs.env e.ty syn.e.subst</i>
	<i>inh.e.subst</i>	= <i>inh.lhs.subst</i>
	<i>inh.b.subst</i>	= <i>syn.e.subst</i>
	<i>syn.lhs.subst</i>	= <i>syn.b.subst</i>
	<i>lhs.errs</i>	= <i>e.errs ++ b.errs</i>

Figure 1.24: DHM with AG rules.

## 1 Introduction

In contrast to the monadic approach in Section 1.2.6, the rules are compositional. The distribution of the environment, the threading of the substitution, and the collection of error messages can be described separately and relatively independently. On the other hand, the rules are also more verbose because the environment, substitution and error messages are not hidden. Furthermore, rules that employ *generalize* are crosscutting as they deal with types, substitutions and environments. This has a negative effect on the degree of separation in the descriptions of these individual attributes. We address both issues in Section 1.3.12.

### 1.3.12 Copy Rules and Collection Attributes

We often pass values in standard top-down, bottom-up, and in-order patterns between attributes of the tree. The rules that encode these patterns are trivial: essentially identity-functions between attributes. For example, to pass an environment topdown to the children of an application we use the following rules:

```
sem Expr prod App  
  inh.left.env = id inh.lhs.env -- copy down left  
  inh.right.env = id inh.lhs.env -- copy down right
```

To thread<sup>13</sup> a counter *uid* (unique identifier) through the tree, we use the following rules:

```
sem Expr prod App  
  inh.left.uid = id inh.lhs.uid -- copy down to left  
  inh.right.uid = id syn.left.uid -- from left to right  
  syn.lhs.uid = id syn.right.uid -- copy up from right
```

Similarly, if a production has one child, we may pass the value of an attribute of that child bottom-up as value for the same attribute of the parent:

```
sem Expr prod Lam syn.lhs.errs = syn.b.errs -- copy error messages up
```

Such *copy rules* [Magnusson et al., 2007] are so common<sup>14</sup> that we allow these rules to be omitted.

To make such an AG well-formed, the following algorithm augments an AG with copy rules. If a rule is missing for an inherited attribute *a* of a child *c*, we insert a rule that takes an attribute with the same name *a* from the local attributes, or synthesized attributes of the children to the left *c*, or the inherited attributes of the parent. The last attribute occurrence is taken in the ordering: *inh* from parent, *syn* of children, local attributes, inherited attrs of parent. In a similar way, we treat omitted copy rules for synthesized attributes. These copy rules can be considered to provide generic behavior for AGs that is not unlike the abstraction offered by reader and state monads.

<sup>13</sup> When we write that we thread an attribute *x* through the tree, then we actually mean that we thread a value through the tree via a sequence of attributes that are all named *x*, and that are chained together by (mostly) copy rules. In a similar way, we talk about passing attributes topdown and bottom up.

<sup>14</sup> The AGs of UHC have more than twice as many copy rules than explicitly written rules. Thus the inference of copy rules saves a lot of manual labor.

Furthermore, we often collect attribute values in a bottom-up fashion:

$$\begin{aligned} \mathbf{sem} \text{ Expr } \mathbf{prod} \text{ App} \quad & \text{syn.lhs.errs} = \text{syn.left.errs} ++ \text{syn.right.errs} \\ \mathbf{sem} \text{ Expr } \mathbf{prod} \text{ Const} \quad & \text{syn.lhs.errs} = [] \end{aligned}$$

Such *collection rules* [Magnusson et al., 2007] can also be inferred automatically when we specify a combination operator and an initial value:

$$\mathbf{attr} \text{ Expr} \quad \mathbf{syn} \text{ errs } \mathbf{use} \ (+) \ []$$

This approach captures the abstraction provided by writer monads.

**Intermediate nodes and copy rules.** It is often convenient to have to have intermediate nodes in the AST structure. An important benefit of copy rules is that it allows us to transparently add intermediate nodes to the AST.

As an example, consider function application in the lambda calculus. It is usually expressed as a binary expression in the abstract syntax:

$$\mathbf{grammar} \text{ Expr} \quad \mathbf{prod} \text{ App} \quad \mathbf{nonterm} \ f, a : \text{Expr}$$

A function call with multiple arguments (e.g.  $f \ a_1 \ a_2$ ) is thus encoded as a sequence of applications ( $(f \ a_1) \ a_2$ ) using *App*. Suppose that we want to add an inherited attribute that describes at which argument position an expression occurs:

$$\begin{aligned} \mathbf{attr} \text{ Expr} \quad & \mathbf{inh} \ \text{index} :: \text{Int} \\ \mathbf{sem} \text{ Expr} \quad \mathbf{prod} \text{ App} \quad & \text{inh.f.index} = \text{inh.lhs.index} \\ & \text{inh.a.index} = 1 + \text{inh.lhs.index} \end{aligned}$$

This definition is incorrect for nested function calls (eg.  $f \ a_1 \ (g \ a_2)$ ), because the first argument  $a_2$  of the nested call receives 2 as value for its *index* attribute. We get the expected behavior by distinguishing whether an expression occurs to the left or right of an application with an inherited attribute. This, however, requires us to specify this attribute for each occurrence of an expression nonterminal.

We obtain a more concise solution if we assume there is always a special top node above a sequence of applications:

$$\begin{aligned} \mathbf{grammar} \text{ Expr} \quad \mathbf{prod} \text{ AppTop} \quad & \mathbf{nonterm} \ e : \text{Expr} \\ \mathbf{sem} \text{ Expr} \quad \mathbf{prod} \text{ AppTop} \quad & \text{inh.e.index} = 0 \end{aligned}$$

The copy rules transparently connect the remaining attributes of  $e$  with attributes of *lhs*. For these attributes, the existence of the intermediate node is not visible. A typical place to add these intermediate nodes is in the parser or with a tree transformation.

If we furthermore ensure that a special root node occurs above expression trees, then we also easily define an initial value for the *index* attribute:

$$\begin{aligned} \mathbf{grammar} \text{ ExprTop} \quad \mathbf{prod} \text{ Top} \quad & \mathbf{nonterm} \ e : \text{Expr} \\ \mathbf{sem} \text{ ExprTop} \quad \mathbf{prod} \text{ Top} \quad & \text{inh.e.index} = 0 \end{aligned}$$

With nonterminal sets (Section 1.3.6) we can define attributes that are common to *Expr* and *ExprTop* without code duplication.

```

grammar Unify prod Unify      -- nonterminal that represents unification
attr Unify  inh ty1, ty2 :: Ty    -- the two input types
                inh env      :: Env    -- the input environment
                chn subst  :: Subst   -- the input/output substitution
                syn errs   :: Errs   -- the output error messages
sem Unify prod Unify          -- essentially a wrapper around unify
      (lhs.errs, syn.lhs.subst) = unify lhs.env lhs.ty1 lhs.ty2 inh.lhs.subst

```

**Figure 1.25:** Encoding of *unify* as a higher-order child.

```

sem Expr prod App
  child u : Unify = sem.Unify          -- higher-order child
  u.ty1      = f.ty                      -- input type
  u.ty2      = (a.ty → loc.res)         -- input type
  inh.u.subst = syn.a.subst              -- copy rule
  syn.lhs.subst = syn.u.subst           -- copy rule
  u.env       = lhs.env                 -- copy rule
  lhs.errs    = f.errs ++ a.errs ++ u.errs -- copy rule

```

**Figure 1.26:** The DHM rules for *App* with copy rules.

**Higher-order children and copy rules.** To improve the effectiveness of copy rules further, and in general to improve the separation of concerns, we can encode crosscutting rules as higher-order children (Section 1.3.7). By encoding a rule *r* as a child, we abstract *r* from how it is combined with other rules.

This idea is the spirit of this thesis. A higher-order child can be used to declaratively specify tasks (for instance, to ensure that two terms are equal), and the underlying implementation (the unification function) reflects the effects of performing the task in terms of attribute values. The orchestration of these tasks is determined by how the rules weave the attributes together.

In general, we can represent any function as a higher-order child. In Figure 1.25 we introduce a nonterminal for unification that has inherited attributes for each input of unification and synthesized attributes for each output of unification. The nonterminal has only one production, which contains only one rule, which is the rule we abstract over.

In a similar way we can encode the *fresh* and *instantiate* functions as higher-order children (Section 5.2.3). In Figure 1.26 we show how this is done for production *App*, where we added the *unify*-nonterminal as higher-order child *u* to the production, and added rules for the attributes of *u*. The latter rules are all implied by the copy rule mechanism and can be

omitted.

**Remarks.** The automatic completion of an AG with copy rules is a double-edged sword. The mechanism saves a lot of boilerplate code, but the automatic behavior may not always be intended. If we accidentally forget to define an attribute explicitly, and that attribute can be given a default definition via a copy rule, then we are not warned that a rule is omitted. We provide a way to specify copy rules per production in Chapter 5, which gives more control over where and what copy rules are applied.

The copy rule mechanism uses the order of appearance of children for threading and bottom-up collections. This is sometimes not the appropriate order for a given application. For example, higher-order children follow conventional children in the order of appearance, thus are always at the end of copy rule chains. In such situations we can override the copy rule behavior by giving explicit rules for attributes. However, this reduces the convenience of copy rules, and the underlying idea that we rather specify patterns than individual rules.

As a solution, in Chapter 3 we allow the visit order to children to be explicitly specified for eagerly evaluated AGs, and can then use copy rules that use the visit-order instead of the order of appearance. Moreover, in Chapter 4 we define commutable (copy) rules, which are rules that can be ordered independently of their value dependencies.

### 1.3.13 Advantages and Disadvantages

The greatest advantage offered by AGs is that specifications are composable. Productions, attributes and rules can all be specified separately and automatically combined into a monolithic specification (Section 1.3.6). This easily allows new attributes and behavior to be added and shared with already existing rules. In particular, extra attributes can be used to specify additional administration for type inference strategies, and for the specification of what a compiler does with the inferred types.

In general, AGs offer modularity [Farrow et al., 1992] and extensibility [Viera et al., 2009], which can be realized via generic programming and meta programming approaches. In this thesis, we make use of such facilities, although these facilities themselves are not the focus of this thesis. Instead, we focus on combining AGs with algorithms that implement the functionality specified by declarative aspects of type rules.

## 1.4 Background on Ruler

This thesis complements previous work by Dijkstra and Swierstra [2006b] on the Ruler language and tool suite. Ruler gives a semantics to type rule descriptions in the form of an implementation with conventional attribute grammars and Haskell. Consequently, Ruler in its current state provides only an implementation for syntax-directed type rules.

The purpose of this thesis is to provide a core language *RulerCore* for Ruler which allows more complex inference strategies to be described. RulerCore extends on attribute grammars in various ways. Therefore, we focus more on attribute grammars than on type rule descriptions, although type systems play a prominent role in our work. The work on Ruler provides

## 1 Introduction

a notation for type rules and show how this notation translates to attribute grammars. From this work a notation that maps to RulerCore can be designed, hence we describe Ruler in this section.

### 1.4.1 Ruler Features

Ruler aims to generate both a type system specification and a type inference implementation from a single description of the type system with type rules. The generated specification consists of type rules formatted to L<sup>A</sup>T<sub>E</sub>X figures, and can be used as documentation and for formal reasoning. The generated implementation consists of attribute grammars that can be included verbatim in the source code of a compiler. This approach guarantees consistency between the specification and the implementation.

Ruler provides notation for the incremental description of type rules. It features the addition of parameters to judgements, type rules to relations, and judgements to type rules in a similar way as we can add attributes, productions, and rules to an AG description.

When type rules are declarative, only well-formedness checks and generation of the specification is possible. However, Ruler provides also notation to describe algorithmic rules. Using the facilities for incremental descriptions, a direction can be given to the parameters of typing relations, and type rules can be associated with productions of an accompanying attribute grammar, which turns the typing relation in a deterministic function for which AG code can be generated.

### 1.4.2 Ruler Concepts

We take the explicitly typed lambda calculus as described in Section 1.2.3 as basis to show the main concepts and syntax<sup>15</sup> of Ruler. In Ruler, a type system description is a composition of views on relations and their rules.

**Definition (View).** In Ruler, a *view* is a named subset of the declared relations, holes, judgements, rules, etc. Each view describes a type system.

We start with a declarative view, which we give the name *D*. Later, we provide also an algorithmic view with the name *A*.

**Definition (Hole).** In Ruler, the parameters of a relation are called *holes*. The parameters are explicitly named and are explicitly typed.

Figure 1.27 shows how to declare a 3-place relation *expr*. The line with *judgespec* specifies a custom notation (meta-grammar) for the *expr* relation in terms of a meta grammar production. Each hole must be uniquely present as meta nonterminal in this meta production.

**Definition (Ruleset).** A *ruleset* is a group of rules in combination with a collection of meta-information, such as a name for the group.

---

<sup>15</sup> We took the freedom to deviate slightly from Ruler's actual syntax in order to have closer correspondence with the notation that we use in this thesis.

```

relation expr view D                -- declares the typing relation expr for D
holes    g :: Gam  e :: Expr  t :: Ty  -- specifies parameters of expr
judgespec g ⊢ e : t                    -- notation for judgments of expr

relation member view D              -- declares the relation member
holes    g :: Gam  x :: String  t :: Ty  -- specifies parameters of member
judgespec (x, t) ∈ g                -- notation for judgments of member

```

Figure 1.27: Examples of declaring relations in Ruler.

```

ruleset theRules relation expr      -- a set of rules for expr

rule e.var view D                    -- rule for the var-case
judge L : member  (x, t) ∈ g      -- premise with the name L

—
judge R : expr    g ⊢ x : t        -- conclusion with the name R

rule e.app view D                    -- rule for the app-case
judge F : expr    g ⊢ f : t1 → t2 -- premise with the name F
judge A : expr    g ⊢ a : t1        -- premise with the name A

—
judge R : expr                          -- alternative syntax for judgment
| g = g                                  -- binds expression g to hole g
| e = f a                                -- binds f a to hole e
| t = t2                                  -- binds t2 to hole t

```

Figure 1.28: Example of Ruler rules.

A relation is either a foreign relation, or it is specified by the rules of some ruleset. Figure 1.28 shows the ruleset *theRules*. A rule is given an explicit name, and zero or more judgments as premises above the line, and one premise under the line. A judgment has an explicit name and corresponds to a relation. The arguments are either bound to the corresponding holes via the custom syntax in a nameless way, or via a generic syntax where each binding is explicitly given.

The language of Ruler expressions consists of meta variables (such as *f* and *a*), and external constants and operators (such as the function arrow). The interpretation of such terms depends on the target language. Denotations can be given for individual symbols and constants, as well as for (saturated) applications:

```

rewrite ag ((t1 :: Ty) → (t2 :: Ty)) = (TyArr t1 t2) :: Ty  -- denotation of application
external TyArr                                     -- identity denotation
rewrite tex ((t1 :: Ty) → (t2 :: Ty)) = (t1 → t2) :: Ty  -- fully saturated appl only

```

## 1 Introduction

**format** `tex`  $\rightarrow$  = "`\rightarrow`" -- denotation of a symbol

Rewrite rules are applied in a bottom-up fashion. The LHS of a rewrite rule specifies a typed pattern to match against a Ruler expression. The RHS must again be a Ruler expression, which then is assumed to have the given explicit type. These types allow the notation to be overloaded. A rewrite rule applies if both the syntax and the types of the LHS matches the actual Ruler expression.

An explicit denotation must be given for foreign relations. The distinction between conventional relations and foreign relations is that the latter is not defined by rules. An explicit denotation to the host language needs to be given for a foreign relation:

**relation** `member` **view** `D` -- `member` is not specified by rules  
**judgeuse** `tex`  $(x, t) \in g$  -- denotation for  $\LaTeX$   
**judgeuse** `ag`  $(Just\ t) = lookup\ x\ g$  -- denotation for AGs with Haskell  
**format** `tex`  $\in$  = "`\in`" -- denotation of a symbol  
**external** `Just lookup` -- identity denotation

The above declarative specification can be typeset to  $\LaTeX$ . To describe an algorithmic version, additional information needs to be added to the relations and rules. We can describe these additions separately by defining a view `A` that extends from `D`:

**viewhierarchy** `D < A` -- partial order on views

In particular, we need to define how the relations are mapped onto nonterminals of an AG, and turn the relations into deterministic functions such that the parameters can be mapped to attributes.

Ruler code does not stand on its own. The generated code for type inferencing is supposed to be used in conjunction with other AG code. An association is specified between relations (schemes) in Ruler and nonterminals in the AG. For example, the relation `expr` is associated with the nonterminal `Expr`. Its type rules are associated with productions of `Expr`. This correspondence is made explicit by annotating the nonterminal declaration with the name of the relation, and the productions with the name of the corresponding rule<sup>16</sup>:

**grammar** `Expr` [`expr`] **view** `A` -- relation `expr` mapped to nonterminal `Expr`  
**prod** `Var` [`e.var`] -- rule `e.var` mapped to production `Var`  
**term** `nm` :: `String`  
**prod** `App` [`e.app`] -- rule `e.app` mapped to production `App`  
**nonterm** `f` : `Expr`  
**nonterm** `a` : `Expr`

Moreover, the holes are mapped attributes. Judgments represent the semantics of a production. Judgments of a foreign relation are translated to AG rules. The other judgements

---

<sup>16</sup> Ruler allows productions to be defined by combining rules. Within the square brackets may not only be a name of a rule, but also an expression that denotes a composition of rules. An example is the left-biased union of two rules.



```

ruleset theRules relation expr           -- extensions to rules for expr
rule e.var view A                       -- rule for the var-case
  —
judge R : expr  g ⊢ (node nm = x) : t -- association between term nm and x
rule e.app view A                       -- rule for the app-case
  —
judge R : expr                             -- convenient for extensions
  | e = (node f = f) (node a = a)       -- assoc child f to f and child a to a

```

**Figure 1.29:** Rules demonstrating node-holes.

correspond to children of the production and are mapped to rules that define the inherited attributes of the children, or pattern match against the synthesized attributes.

In view *A*, we refine the declaration of *expr* to include additional information. In this example, we only provide additional information about the attributes. In general also additional attributes and syntax can be added. We map the holes of the *expr* relation to inherited or synthesized attributes. The hole that corresponds to the AST gets the special *node* designation:

```

relation expr view A   -- algorithmic view on expr
holes node e :: Expr  -- AST node
inh g :: Gam          -- input param is inherited attribute
syn t :: Ty           -- output param is synthesized attribute

```

A Ruler expression is at a defining position if it is bound to an inherited hole of a premise, or bound to a synthesized hole of a conclusion judgment. Otherwise, it is at a using position. The generated algorithm constructs a value at defining positions, and matches against values at usage positions.

A node-hole takes the AST as value. In judgments of rules, node holes are treated differently with respect to normal holes in order to define the correspondence between judgments of the type rule and children of the production. The node hole of a premise judgment may only be an identifier and corresponds uniquely to child of the associated production. In the expression bound to the node hole of the conclusion, these identifiers must occur and we annotate them with the name of the child of the production. Figure 1.29 shows an example. The syntax for explicitly named bindings of holes is convenient for extensions, as only the bindings that are redefined need to be mentioned.

When a Ruler description is well-formed, an AG can be generated from the above description. This AG contains attributes and semantics for type inference, and can be combined with other attributes and Haskell infrastructure into a compiler.

```

viewhierarchy  $D < A < H$            -- DHM view  $H$ 
relation expr view  $H$            -- DHM view on expr
  holes chn  $s :: Subst$          -- substitution as threaded attribute
relation tyFresh view  $H$        -- wrapper for fresh
  holes chn  $s :: Subst$          -- threaded subst
  syn  $t :: Ty$                    -- fresh type
  judgespec  $t \text{ 'fresh'}$          -- syntax for the judgement
  judgeuse ag  $(t, syn.s) = fresh\ inh.s$  -- denotation for AGs with Haskell
relation tyUnify view  $H$        -- wrapper for unify
  holes chn  $s :: Subst$          -- threaded subst
  inh  $t_1 :: Ty1$                  -- left type
  inh  $t_2 :: Ty2$                  -- right type
  judgespec  $t_1 \equiv t_2$        -- syntax for the judgement
  judgeuse ag  $(retain\ errs, syn.s) = unify\ g\ t_1\ t_2\ inh.s$ 

```

Figure 1.30: DHM relations in Ruler.

### 1.4.3 Damas-Hindley-Milner Inference

Similar to Section 1.2.6 and Section 1.3.11 we show in this section a DHM inference algorithm in Ruler. In the DHM view, we add lambda abstractions without an explicit type annotation and a let expression. However, we only show the code for the app-rule, since we already explained Ruler’s concepts in the previous section.

In Figure 1.30 we add a DHM view on top of the algorithmic view, and define external relations to obtain fresh types and to unify two types. The `retain`-keyword declares that the left component of the output tuple of `unify` is mapped to a local AG attribute `loc.errs`, which can then be collected by conventional AG rules. This mechanism allows values to be exposed as attributes to the encapsulating AG.

Figure 1.31 shows the rules for `e.app` in view  $H$ . The hole-bindings for judgments essentially specify the threading of the substitution. The notation for hole-binding can be used to supply bindings for holes that are not present in the judgement’s special syntax. To connect two nodes, we introduce an intermediate meta variables  $s_1, s_2$ , etc. This threading has to be done manually as Ruler does not have a concept of copy rules.

### 1.4.4 Discussion

Ruler provides notation and composition mechanisms for the description of type rules. Rules may inherit from other rules, and rules inherited from the same rule from preceding views. These mechanisms enhance modularity and reuse. Effectively, a ruler fragment is a partial specification. The meaning of a ruler specification is only defined for a complete composition in combination with the associated attribute grammar. It would aid formal reasoning if a

```

ruleset theRules relation expr
rule e.app view H
  judge T : tyFresh r 'fresh'
    | inh.s = s1 | syn.s = s2
  judge F : expr
    | t = t1 | inh.s = s2 | syn.s = s3
  judge A : expr
    | t = t2 | inh.s = s3 | syn.s = s4
  judge U : tyUnify t1 (t2 → r)
    | inh.s = s4 | syn.s = s5
  —
  judge R : expr | inh.s = s1 | syn.s = s5

```

Figure 1.31: Ruler rules for *e.app*

meaning can be attached to individual fragments.

The Ruler compiler generates only an inference algorithm for algorithmic specifications. As a consequence, the code generation is limited to syntax-directed type rules. Syntax-directedness does not hold for many declarative type systems that have relations with overlapping rules, or rules that dispatch on more than one argument of the conclusion judgment. Also, the premisses must be functional. The transformation of a relation to a function by itself is non-trivial, and common techniques such as fixpoint iteration or search strategies are not directly supported by Ruler. We address these complications in this thesis (Section 1.5).

Moreover, Ruler's features are actually not specific to the domain of type systems. Ruler provides a rudimentary composition mechanism, syntactic sugar, and type setting support for a formalism that is not unlike AGs. These features would equally well benefit AGs in general<sup>17</sup>.

## 1.5 Thesis Overview

Our ultimate goal is to semi-automatically generate a type inference algorithm from a declarative type system specification. In particular, we focus on type systems described as a collection of type rules, and an implementation based on attribute grammars.

In this thesis, we present RulerCore, a language that extends attribute grammars. It facilitates the composable description of inference algorithms that are typically used to implement declarative aspects of type rules. RulerCore can thus be used to give an executable semantics to a set of type rules.

<sup>17</sup> An example of a feature that benefits AGs is Ruler's automatic unique numbering mechanism. We generalized and implemented a similar mechanism for AGs.

In this section, we give an overview of RulerCore’s extensions on attribute grammars. In Section 1.6 we position this thesis with respect to the larger goal.

### 1.5.1 Inference Algorithms as an Attribute Grammar

We give an abstract description of how we structure inference algorithms as an AG. Chapter 5 shows a concrete example.

We give AGs over typing derivations, instead of AGs over the abstract syntax of a language. In the following chapters of this thesis, we refer with *abstract syntax tree* either to typing derivations or to the result of parsing<sup>18</sup>. Nonterminals thus correspond loosely to typing relations, and productions to type rules (Section 1.3.10). The grammar for typing derivations may have more structure than is present in the type rules. For example, we may add nodes that each represents a choice between alternatives, so that we effectively describe a forest of typing derivations.

We map each declarative aspect of a type rule to a higher-order child (Section 1.3.7) as shown by Section 1.3.12. Extra attributes, such as a substitution, provide contextual information for these children. For example, an equality premise between two types in the type rules corresponds to a unification-child in the AG. The structure of the unification child serves as proof that the two types can be made equal, and the resulting substitution attribute reflects the effect of unifying the two types. We treat meta variables in type rules as conventional attribute values (Section 1.3.11).

Type inference amounts to determining the structure of these children. To choose a particular tree for a child may require an exploration of candidate trees. Instead of constructing a single derivation tree, we actually construct and choose from a forest of derivation trees. Through value dependencies between attributes, we effectively define in which order the structure of these children is determined, and in which order the effects of determining this structure is visible in the substitution attribute.

This approach allows us to encode Algorithm W (Section 1.2.6). For more complex inference algorithms, such as constrained-based algorithms and algorithms that require fixpoint iteration, the shape of the derivation tree and the values of attributes are mutually dependent. Inference algorithms therefore analyze, explore, and extend (intermediate) partial derivation trees. This process does not have a straightforward mapping to an AG, since an AG specifies when the resulting derivation tree is correct, but not how the intermediate trees are obtained. We introduce extensions to AGs to make such intermediate trees visible in the AG description.

### 1.5.2 Attribute Grammar Extensions

Chapter 2 gives a detailed outline of each extension. The following chapters work out each extension individually.

---

<sup>18</sup> The typing derivation is typically an extension of the AST, thus the difference is usually irrelevant and can be determined from the context. Similarly, we refer to AST and the AST decorated with attributes interchangeably.

**Visits.** RulerCore is a language for the description of higher-order, ordered attribute grammars. We extend this basis with explicit specifications of visits. In comparison, visits are implicit in ordered attribute grammars. Chapter 3 introduces the language and the notation.

For ordered attribute grammars, there exists an evaluation algorithm that starts with an initially undecorated tree and ends in a correctly decorated tree. The *state* of a tree is the collection of decorations present in the tree. During the evaluation, the state of the tree thus changes. A *configuration* is a set of attribute names. A configuration describes the state of a tree when the set of decorations of the root contain exactly the attributes as mentioned in the configuration. In an ordered attribute grammar a linear order exists between configurations. A *visit*, a unit of evaluation for a node, transitions the state of a tree to a state described by the next configuration. We treat a node with a state described by a certain configuration as a *first class value*, which we can store in attributes, obtain from attributes, inspect, and programmatically apply state transitions on. This approach offers us sufficient control over the AG evaluation to combine AGs with *monadic operations*, such as the unification monad as shown in this chapter.

We provide notation to declare a totally ordered sequence of visits per nonterminal. Each attribute declaration must be associated with one visit declaration, which has consequences for the scheduling of rules. During attribute evaluation, attributes that are associated with an earlier visits are defined before attributes of a later visit are computed/defined. In this context *defined* means that a reference to the attribute value is available. Similarly, rules of productions of a nonterminal may be associated with a visit of that nonterminal, which restricts the scheduling of such rules to either that visit or to a later visit. Furthermore, we may restrict a rule to a particular visit, which ensures that the rule is scheduled before any rules of subsequent visits.

With this approach, rules may make assumptions about the configuration of tree tree prior to the rule's evaluation. Moreover, the notation allows us to define customizations of evaluation strategies for rules of a particular visit or for visits to particular subtrees.

A disadvantage of our approach with respect to attribute grammars is that we need to specify a visit for each attribute. This requires additional effort and makes attribute declarations less composable. We typically declare an attribute for a set of nonterminals instead of a single nonterminal. In our approach, this is only possible if all the nonterminals in the set have a common visit to which the attribute can be scheduled. We show in Chapter 4 how to solve this issue.

**Fixpoint iteration, clauses and constraints.** In Chapter 5 we exploit the notion of visits. We show how we to conditionally repeat the evaluation of a visit to a child, which allows the encoding of *fixpoint iteration*. Using visit-local attributes, a state can be kept between iterations. Moreover, we allow one or more clauses to be defined for a visit. Each clause provides an alternative set of rules for the visit. With special match-rules we specify constraints on clauses. With clauses in combination with higher-order children we can define the structure of the derivation tree in terms of attributes, and thus deal with type-directed inference algorithms.

We treat intermediate derivation trees as first class values. With the specification of visits, we can reason about the configuration a tree is in. A tree that is in a certain configuration

## 1 Introduction

can be detached, transferred via attributes to another location, and attached there. With this mechanism we can represent *constraints* or *deferred judgments* as trees with access to their context via attributes.

**Exploration of alternatives.** The above extensions describe algorithms that conservatively approximate the derivation tree. For some type systems it is necessary to explore a forest of candidate derivation trees. Such a forest can be represented with a decision tree, which contains choice nodes that branch to various *alternatives*.

In Chapter 6 we show how to describe explorations of such alternatives with AGs. We present a technique that allows a spectrum of depth-first and breadth-first search strategies to be described. In a statically scheduled AG, we can evaluate the AG in a step-by-step fashion. By intertwining the evaluation of alternatives, we obtain a breadth-first search. After each step, some intermediate values may be available, which can be used to direct the search process.

**Phases and Commuting Rules.** In Chapter 4 we generalize visits to *phases*. A phase may consist of one or more implicitly defined visits, which are determined by the static scheduling of the AG. As a consequence, an attribute does not need to be explicitly assigned to a visit, and its scheduling may optionally be constrained by a phase. Using this approach, our extensions extend AGs conservatively.

For some chained attributes, the order induced by value dependencies of their rules may be too strict when these rules encode commuting operations. We present *commuting rules*, which are rules that are connected via a chained attribute, but which do not depend on previous rules in the chain. Such rules thus give us more freedom in the scheduling of these rules. Typical examples are the threading of a unique number supply, and the threading of substitutions. To preserve referential transparency, the commuting rules must satisfy a liberal commutativity law. With such rules we can functionally encode the behavior of a rule with side effect that is scheduled to different implicit visits.

**Dependent AGs.** In Chapter 7 we apply dependent types to AGs. In a dependently typed AG, the type of an attribute may refer to values of attributes. The type of an attribute is an invariant, the value of an attribute a proof for that invariant. Thus, with dependent AGs we can proof properties of our compiler. Additionally, this chapter serves as a showcase for visits and clauses.

### 1.5.3 Contextual Chapters

In the extended edition of this thesis<sup>19</sup>, we place the above extensions in a wider context.

---

<sup>19</sup> Extended edition: <https://svn.science.uu.nl/repos/project.ruler.papers/archive/thesis-extended.pdf>

**Graph Traversals.** Many computations in a compiler take control-flow or data-flow graphs into account. We show that the mechanism to attach and detach children can be used to interface AGs with graph traversals [Middelkoop, 2011b].

**GADTs.** In the extended edition [Middelkoop, 2011a], we show an example of a type system for Generalized Algebraic Data Types (GADTs). An inference algorithm for this system requires an exploration of alternatives (Chapter 6). We formulate our specification so that it is orthogonal to specifications of ADTs:

- We present our specification as System F augmented with first-class equality proofs.
- We exploit the Church encoding of data types to describe GADT matches in terms of conventional lambda abstractions.

Such orthogonal designs are important in order to compose type systems, and ultimately thus also to compose type inference algorithms.

## 1.6 The Context of this Thesis

Types play an increasingly more important role in the design of programming languages. Type systems specify a relation between programs and types, which facilitates (formal) reasoning with typed programs. Moreover, type systems form a partial specification for type checking and type inference algorithms. As we discussed in Section 1.1, our ultimate goal is to semi-automatically derive type inference algorithms from declarative type system specifications. We mentioned in Section 1.2 that a set of type rules alone is not a complete description, hence we develop Ruler (Section 1.4), which is a domain-specific programming language in which we write an inference algorithm as an extension of the declarative type rules.

### 1.6.1 Challenges

There are several challenges that need to be overcome to reach this goal. We identify two main challenges. This thesis is situated in the second challenge.

The first challenge is related to type system compositions. Language features and their declarative type systems are typically defined as extensions of a bare lambda calculus. Some language features are mutually conflicting (e.g. invalidate type soundness). However, many language features compose in standard ways. For example, features described for a lambda calculus that support fix-expressions can be translated to a description for a language with recursive let bindings.

To meet this challenge, we wish to describe language features in isolation, and describe a composition of these features for the actual source language. As illustration, the type rule formalism lacks the expressiveness that higher-order functions offer to functional programs, such as the ability to abstract over common patterns, and to instantiate these abstractions with minimal syntax. For small type system descriptions that appear in type system theory,

## 1 Introduction

such expressiveness is not needed. However, type system descriptions of actual languages are large and hard to maintain.

The second challenge is related to declarative type rules. Declarative type rules abstract from evaluation strategies. However, a general inference algorithm does not exist, and a naive algorithm such as mentioned in Section 1.2.4 is either incomplete or inefficient.

Ultimately, inference boils down to resolving *declarative aspects*: to determine the structure of the derivation tree, and computing with values that may not be fully determined yet. In practice, inference algorithms are intricate compositions of common algorithms that treat such declarative aspects in a predictable and deterministic way. These algorithms are hard to combine. If one declarative aspect requires a constraint-based algorithm for its resolution, and another requires some form of search, then the order in which the aspects are resolved is likely to be relevant. Also, it is hard to describe the flow of information between different solving techniques. To allow these techniques to mutually cooperate, we need a language for the description of a composition of such techniques. Hence, this thesis.

### 1.6.2 Additional Challenges

Aside from the general motivation of our research, another source of motivation is that we intend the results of our research to benefit the implementation of our Haskell compiler UHC [Dijkstra et al., 2009]. Therefore, we impose additional demands on solutions to the above challenges.

Firstly, since UHC’s implementation is based on Haskell itself, we require that solutions integrate seamlessly with Haskell. This restriction effectively rules out the direct use of (functional) logic languages, due to differences in the evaluation model and the representation of data structures<sup>20</sup>. In addition, we desire that our research can also be exploited in compiler suites that are implemented with languages without lazy evaluation or strict typing disciplines.

Secondly, along similar lines, we refrain from the use of dependently-typed languages, since an extraction to Haskell is a one-way process that also affects data-type representations. Our goal is to be able to generate an implementation. Advances in dependently-typed languages seem promising, but a formally certified implementation of a compiler such as UHC is currently infeasible.

Finally, the resulting implementation should be reasonably efficient in order to process ASTs of large programs. In our experiences with UHC, we noticed that memory usage is an issue when using demand-driven evaluation of AGs. We experimentally verified that the time spend on traversing abstract syntax trees in UHC is negligible in comparison to the computations that are performed on each node of the AST. Thus, while traversal overhead is rarely a problem, memory usage is an item of concern, which we address by using statically ordered evaluation of AGs.

---

<sup>20</sup> Braßel et al. [2010] show that an embedding of functional logic programs is possible in Haskell, but affects all data representations and forces all computations to monadic style. However, we use techniques techniques and ideas from logic programming that integrate seamlessly, such as backtracking in a monad [Hinze, 2000, Kiselyov et al., 2005].



### 1.6.3 Solutions

A partial solution to the first challenge is given by Dijkstra [2005], as demonstrated by the UHC project, and the initial development of Ruler (Section 1.4) in particular.

Ruler’s composition mechanisms and syntax extensions that are provided by Ruler would be beneficial to AGs. For example, many dense translation schemes in this thesis are manually derived from actual AG descriptions. These AG descriptions focus on attributes in isolation and are easier to understand, but too verbose for inclusion in this thesis. Solutions for AGs would also work for type rule descriptions, and vice versa. Indeed, composition facilities for AGs receive ongoing attention [Viera et al., 2009, Saraiva, 2002].

There is still a long way to go with respect to the first challenge. Since declarative rules abstract from an evaluation algorithm, the data structures and administration that are involved in the algorithm are chosen for convenience and notational conciseness. In an actual implementation, we may be able to represent certain administration in a more efficient way using specialized data structures. However, we do not address these issues in this thesis, and only mention some in passing in Middelkoop [2011a]. We assume that the declarative rules are specially crafted to make them more suitable for an actual implementation.

There is thus some open work for the first challenge. However we focus on the second challenge. This challenge is more pressing, because we need basic building blocks before we can compose them in clever ways.

We propose to tackle the second challenge with attribute grammars. Inference algorithms that are specified by declarative type rules are sensitive to context—non-inductive properties of the AST—and attribute grammars excel in providing such contextual information with attributes, as is proven by UHC’s implementation. Also, from a practical perspective, since UHC’s implementation is based on AGs, an inference algorithm based on AGs interfaces conveniently with attributes of other components in the compiler, as shown by previous work on Ruler.

However, attribute grammars in current form are not well suited for the description of inference algorithms of complex type systems. Inference algorithms therefore make explicit assumptions about the intermediate states of the derivation trees during its construction. In a conventional AG, we cannot do so, because AG descriptions are defined in terms of the final state of the derivation tree. In order to make assumptions about the intermediate state, we extend AG evaluation and hence arrive at Section 1.5.

Our approach applies to the description of algorithms that are recursive functions over tree-like data structures. In particular, our approach applies to catamorphisms, which is not surprising because attribute grammars can be considered a domain specific language for the description of catamorphisms. On the other hand, for example, algorithms based on graph rewriting are not straightforwardly expressed in our approach. An inherent difference is that we traverse a structure whereas rewrite rules as used by graph rewriting access the structure in irregular ways. Also, algorithms that involve matrix operations to efficiently solve linear constraints cannot be described straightforwardly. However, Middelkoop [2011b] shows how to mix attribute evaluation with external solvers.

## 1.7 Related Work

Each chapter has its own related work section. In this section we consider work that is related to the thesis as a whole.

### 1.7.1 Circular AGs and Exposure of Intermediate States

Evaluation algorithms for circular AGs [Jones, 1990, Magnusson and Hedin, 2007] provide an alternative way to extend the AG evaluation algorithm. For circular AGs, the algorithm is parametrized with an initial value for cyclic attributes, which are called gate attributes by Walz [1989]. The algorithm describes a repeated attribute evaluation to compute a fixpoint for the cyclic attributes. When used in the context of this thesis, such attributes thus expose intermediate states of the derivation tree during evaluation.

Fixpoint iteration is one of many evaluation strategies (Section 1.2.4). For example, several type systems use an algorithm that describes the exploration of multiple candidate derivation trees (Chapter 6). Moreover, to expose the intermediate states of the derivation tree as an attribute, attributes at various locations of the tree have to be explicitly stored into and obtained from this attribute<sup>21</sup>, which is cumbersome and destroys modularity. Our extensions generalize over fixpoint iteration and many other common techniques employed by inference algorithms.

### 1.7.2 Inference Rules

Glesner and Zimmermann [1998] shows a mapping from syntax directed inference rules to AGs. Similar to Ruler, relations defined by syntax directed rules are mapped to nonterminals, syntax directed rules to productions, and judgments of such relations to children of a production. Judgments of rules that are not syntax directed are mapped to an AG rule that employs a hull algorithm to construct a derivation tree. Only one of the rules may be applicable at a given time, and the domain of the parameters of the relation must be finite. While suitable for type checking, this approach does not suffice for type inference, as multiple rules may be applicable, and it may be necessary to postpone solving the judgment when insufficient type information is available.

Typol [Despeyroux, 1988] is another language for the specification of inference rules with a mapping to Prolog. Attali and Parigot [1994] show how to identify a subclass of Typol that can be mapped to an efficient AG implementation. Since this class consists effectively of syntax directed rules, this approach is too restrictive for the implementation of type inference.

### 1.7.3 Proof Assistants

Proof assistants such as Isabelle/HOL [Wenzel et al., 2008], Twelf [Schürmann, 2009], and Sparkle [Mol et al., 2002] and Coq [Bertot, 2006] can be used to formalize a type system and inference algorithm, and prove various consistency properties between specification and implementation. Typically, the formalized inference algorithm can be extracted to code in

---

<sup>21</sup>Such an attribute models a heap in a similar way as a substitution models memory.

some target language. Such an approach is possible for small type systems as encountered in theory, but does not scale up when dealing with practical, full-blown type systems of large languages<sup>22</sup>.

In Coq, but also in other dependently typed languages such as Agda [Norell, 2009], Epigram [McBride, 2004], and IDRIS [Brady, 2011], properties of type system can be expressed as types of the inference algorithm. To do so, the inference algorithm needs to be implemented, and structured so that it can be complemented with proofs of properties, such as type soundness. In this thesis, we consider type systems for which the first task is already difficult, and the second task infeasible in practice. Thus, such approaches are out of the scope of this thesis. However, we consider dependently typed languages in Chapter 7.

Closer to the goals expressed in this thesis are the languages Ott (Section 1.7.4) and TinkerType (Section 1.7.5).

### 1.7.4 Ott

Ott [Sewell et al., 2007] and SASyLF [Aldrich et al., 2008] are meta languages in which formal semantics such as type systems can be formalized. Similar to Ruler [Dijkstra and Swierstra, 2006b], these languages provide special syntax for inference rules, and require the rules to be well-formed. In contrast to Ruler, the purpose of these languages is to aid the construction of proofs. From an Ott-description, boilerplate code for several proof assistants (e.g., Coq) can be generated. This boilerplate code consists of parsers for concrete syntax, abstract syntax, and substitution lemmas.

**Concrete and Abstract Syntax.** The following is specification of a grammar for a variant of the lambda calculus with tuples in Ott<sup>23</sup>. Such a grammar consists of three types of symbols: meta variables, nonterminals and terminals. Meta variables are nonterminals that can be substituted and alpha-renamed. Nonterminals are introduced by the grammar. The remaining symbols that occur in the grammar are considered terminals. Occurrences of a nonterminal may take a subscript to distinguish multiple occurrences of the same nonterminal from each other.

```

metavar x
grammar e ::=                               :: Expr
        | x                                 :: Var
        | \ p . e                           :: Lam :: bind b(p) in e
        | e1 e2                             :: App
        | ( e )                             :: _

        p ::=                               :: Pat
        | x                                 :: Var :: b = {x}
        | (p1, ..., p.n)                   :: Tup :: b = b(p1) ... b(p.n)

```

<sup>22</sup> As demonstrated by Faxén [2002], a type system for Haskell'98 is already large and complex.

<sup>23</sup> For presentation purposes, the examples use a slightly different notation than provided by Ott.

## 1 Introduction

The grammar specifies a concrete syntax to the left of the double colon, and the constructor for the abstract syntax to the right. The production for parentheses is considered a meta-production and is not reflected in the abstract syntax.

An important concept of Ott is *binding*. After the second double colon, *binders* for meta variables can be specified, as well as their scope. The expression  $b(p)$  represents the set of meta variables defined by the synthesized attribute  $b$  of  $p$ . Zero or more synthesized attributes may be specified for a nonterminal. The binder  $\text{bind } b(p) \text{ in } e$  for the lambda production denotes that the meta variables  $b(p)$  are bound at this lambda, and are in scope of  $e$ . Lemmas for substitution and alpha equivalence, as well as a definition of free variables are derived from the binder annotations. The underlying mechanism ensures through alpha renaming that meta variables are not accidentally captured by substitutions.

Ott has a notion of list forms, which is convenient syntax to represent the common over-bar notation that is used grammars and type rules. Triple dots can be used to construct list patterns, list expressions, and lists of judgments. For example  $b(p1) \dots b(p.n)$  is a list expression, where  $n$  is an index variable. Also, list comprehensions and projections of list items can be used.

**Inference Rules.** Judgements can refer to relations defined in Ott via type rules, or to externally defined functions and relations in the target language. For example, to model a call-by-value operational semantics, the following code fragment represents beta reduction.

```
defn e1 --> e2 :: reduce
      isValue e2
----- :: call-by-value beta
      (\x.e1) e2 --> {e2/x} e1
```

The `defn` line specifies the syntax of the `reduce` judgments, and is followed by the inference rules for that relation. The premise judgments occur above the horizontal line and conclusion judgments below. The infrastructure for the substitution  $e2/x$  is derived from the binder specification of  $e1$ . The inference rules are translated to axioms in the target language for the reduction relation.

**Discussion.** Approaches such as Ott aid formal reasoning about type systems, and thus pursue a different goal than the derivation of type-inference implementations from specifications. On the other hand, Ott and Ruler share the common goal of formalizing type systems and specifying properties. Certain concepts are also beneficial to AGs. Binding and scoping is very common, so the concept of binding may be very useful for AGs as an abstraction for name analysis. Similarly, list forms would benefit AGs when using higher-order children (Section 1.3.7).

### 1.7.5 TinkerType

TinkerType [Levin and Pierce, 2003] is a language for the modular description of whole families of formal systems, with a focus on type systems and operational semantics. A type

system is described in two ways. A system is described intensionally as a set of *features*. These features are names for abstract properties of a type system. A system is described intensionally as a set of *clauses*. In TinkerType, a clause is a denotation of a type rule in the form of  $\LaTeX$  text or ML code.

**Overview.** A TinkerType description consists of a number of elements: features, dependencies between features, clauses, a clause refinement relation, and feature constraint formulas. With the latter two elements, clause refinement and feature constraints, a partial consistency between type systems can be expressed.

Distinct type systems have different clauses. However, type systems with similar features tend to have similar clauses. The relation between clauses is exploited by TinkerType. A TinkerType description therefore contains a whole repository of named clauses that are tagged with a number of feature names. Clauses with different feature sets can have the same name. Given a number of features, a type system is then assembled by the TinkerType Assembler by selecting the clauses that support these features best, which are the clauses where the provided feature set is a subset of the requested feature set. Duplicately tagged clauses are filtered out. The clause with the largest feature set is retained.

A dependency relation must be specified between features. A type system is fully defined by the transitive closure of the dependency relation on the set of features of the type system. Some combinations of features give an unsound type system, or the inference algorithm is incomplete. Certain combinations of features can be declared as deficient using feature constraint formulas. These are propositional formulas over features that must be satisfied with the features mapped to truth-values based on their presence in the type system. The dependency relation between features is one form of such a feature constraint formula, in the form of an implication.

**Code Assembly.** The code repository consists of *components*, which represents several clauses and support code. For example, the following fragment [Levin and Pierce, 2003] contains ML code that deals with the type checking of conditional and boolean expressions.

```

component bool, typing {
  parsing { ... }
  ast { ... }
  core {
    typeof {
      header {# let rec typeof gam t = match t with #}
      separator {#| #}

      T-If
      {#TmIf(e1,e2,e3) ->
        if equiv gam (typeof gam e1} TyBool
        then let res = typeof gam e2 in
          if equiv res (typeof gam e3)
          then res
      }
    }
  }
}

```

## 1 Introduction

```
        else error "branches differ in type"
      else error "guard is not a boolean"#}
    } } }
```

The level of granularity of features is actually per component instead of per clause. A component consists of several sections related to parsing, abstract syntax tree representation, and the actual typing relations with their clauses.

The assembling process is essentially based on concatenating and substituting strings. The components that match the requested features are merged, and the produced code consists of the header followed by the clauses in verbatim, which are separated by the separator.

Consider a component with subtyping `sub` as additional feature:

```
component bool, typing, sub {
  core {
    typeof {
      T-If
      {#TmIf(e1,e2,e3) ->
        if [[subtype]] gam (typeof gam e1) TyBool
        then [[join (typeof gam e2) (typeof gam e3)]]
        else error "guard is not a boolean"#}
    } } }
}
```

In another component with feature `sub`, the functions `subtype` and `join` are defined, which can thus be used in the above component.

The refinement relation between clauses is expressed by means of double bracket annotations in the source code. The fragments inside the double brackets are considered new, the fragments outside the double brackets must occur verbatim in the refined clause.

**Discussion.** TinkerType is modular in the sense that all clauses can be written separately, and the system enforces that clauses are designed with reuse of concepts in mind. A clause can only be reused verbatim. In practice, additional material needs to be added to a clause, for example, due to extra judgments or due to extra parameters to the judgments when supporting extra features. This leads to code duplication in the clauses with the usual engineering problems as a consequence. The static checks on clause refinement, however, are likely to catch errors resulting from incomplete code modifications, and encourage writing clauses as increments of each other.

As discussed above, some forms of consistency are expressed between clauses of different type systems, which is enforced by means of consistency checks that point to errors in the code. To make these checks effective, there must be a lot of overlap between clauses, and thus an extensive set of features with a fine granularity. Consistency is not expressed between clauses of the same type system. There is no guarantee that the collection of clauses results in compilable ML code or  $\LaTeX$  text, nor that the ML code is in any way related to the  $\LaTeX$  text.

## 1.7.6 Overview of Recent Attribute Grammar Systems

An in-depth exploration of AG systems is out of the scope of this thesis. We give some of the distinguishing features of current AG systems. Most current AG systems support a wide range of features including higher-order attributes [Vogt et al., 1989] and collection attributes [Magnusson et al., 2007].

The Lrc [Kuiper and Saraiva, 1998] is one of the few current AG systems that is based on ordered attribute evaluation. Its distinguishing feature is incremental evaluation. It has been used as vehicle for research in parallel evaluation and the generation of interactive programming environments. Lrc generates Haskell and C code, although it is not actively maintained anymore.

UUAG [Löh et al., 1998] can be regarded as a simplified reimplementaion of Lrc, starting originally by piggybacking heavily Haskell's lazy evaluation. Currently, UUAG supports ordered and demand-driven evaluation. Features such as incremental and parallel evaluation are being investigated, as well as first-class attribute grammars [Viera et al., 2009].

The AG systems that we consider below are based on demand-driven attribute evaluation. These systems support reference attributes [Magnusson and Hedin, 2007] which allow attributes to be defined and accessed from non-local nodes. Data-flow analyses with circular attributes are an application of reference attributes [Farrow, 1986].

Silver [Van Wyk et al., 2008] supports forwarding [Van Wyk et al., 2002] as distinguishing feature. Forwarding is a convenient notation for desugaring with higher-order children (Section 1.3.7) in combination with specialized copy rules (Section 1.3.12). With first-class AGs, Viera et al. [2009] implement a more advanced form of forwarding.

Silver also supports the specification of a (control-flow) graph structure on top of the AST [Van Wyk and Krishnan, 2007]. A production may specify CTL formula which are checked against the graph structure. This way, control-flow analyses can be implemented conveniently in attribute grammars (see also Middelkoop [2011b]).

JastAdd [Ekman and Hedin, 2007] has rewrite rules as distinguishing feature. Rewrite rules are applied to a tree upon the first access through demand-driven evaluation and can conditionally depend on attribute values.

## 1.8 Conclusion

In the following chapters, we present several extensions to attribute grammars that facilitate the description of complex type inference algorithms. The central concept in these chapters is that we exploit the explicit notion of visits to control and manipulate chunks of AG evaluation. It allows us to transform the tree during attribute evaluation—precisely what we need to express type inference algorithms. Many classic AG approaches use a notion of visits in their intermediate languages (Section 1.3.4). In this thesis, we instead propose to use visits as *programming model*. With this programming model, we express resolution strategies for declarative aspects of type rules.

Our extensions offer flexible ways to *tune* conventional attribute grammar evaluation, and are conservative extensions of (ordered) attribute grammars. We offer a delicate balance

## 1 Introduction

between on the one hand the implicit evaluation strategy of attribute grammars, and on the other hand the need to make this explicit for more complex evaluation strategies.

Underlying our extensions are well-defined concepts from higher-order AGs (first-class children), and ordered AGs (visits). Underlying these concepts are well-defined concepts from functional programming languages (first-class functions, coroutines, and referential transparency). These concepts form a solid theoretical basis to build upon.

The extensions that we present are not limited to type inference. In fact, type inference is a use case that sets challenges whose solutions improve the abstraction facilities that are available to structure compilers.

**Thesis organization.** Chapter 2 gives a detailed summary of the thesis. Subsequent chapters focus on individual extensions, and reintroduce relevant terminology. For background information related to type systems and attribute grammars, Section 1.2 and Section 1.3 can be used as reference.

**Publications.** The chapters of thesis are based on the following publications:

- We presented an earlier version of Chapter 3 at the Workshop on Generative Technologies (WGT '10) at ETAPS in 2010 [Middelkoop et al., 2010d]. An extended version appeared in the journal of Higher-Order Symbolic Computation [Middelkoop et al., 2011a].
- Some of the work of Chapter 4 is to appear on PADL'12.
- Chapter 5 is an extended version of the paper that we presented at the conference on Generative Programming and Component Engineering (GPCE '10) in 2010 [Middelkoop et al., 2010a].
- An earlier version of the chapter about GADTs (in the extended edition of this thesis) appeared in the post proceedings of Trends in Functional Programming (TFP '08) [Middelkoop et al., 2008], and a later version appeared in the journal of Higher-Order Symbolic Computation [Middelkoop et al., 2011b].
- Chapter 7 is to appear in the post proceedings of the symposium on Implementation and Application of Functional Languages.
- We presented Chapter 6 at the workshop on Language Descriptions Tools and Applications (LDTA '11) in 2011.

As a formal detail, the Association for Computing Machinery (ACM) has copyright on the paper version of Chapter 5 and Chapter 6. Elsevier has copyright on the paper version of Chapter 3.



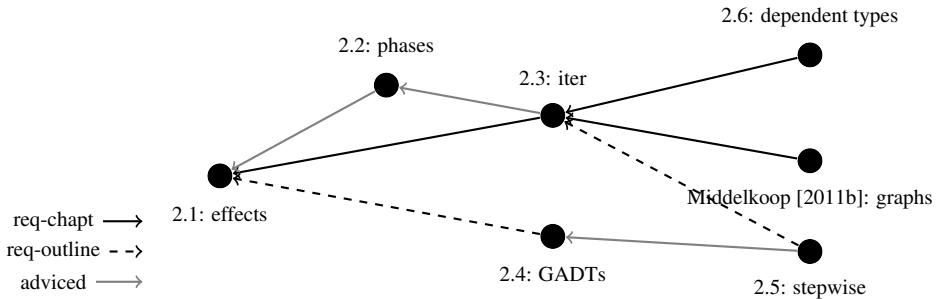
## 2 Outline of the RulerCore Concepts

Section 1.5 argued the necessity of extensions to attribute grammars. In the following chapters of this thesis we describe individual extensions to attribute grammars. In this chapter, we present the language RulerCore and give a detailed summary of the extensions. Each section summarizes a chapter in this thesis.

This chapter can be read before or after the other chapters. It shows how the individual chapters are connected together. This chapter uses a uniform notation, whereas in the individual chapters, we use minor differences in notation when that is more suited for that chapter. Consult the actual chapters for a more extensive explanation and technical material.

**Outline.** Chapter 3 and Chapter 5 give a detailed description of RulerCore’s syntax. In this chapter, we use the syntax as described in Section 1.3.1. Prerequisite to this chapter are ordered attribute grammars (Section 1.3.4) and higher-order attribute grammars (Section 1.3.7).

The following dependency graph shows the dependencies between sections of this chapter (and the corresponding chapters). The solid arrows represent dependencies implied by the contents of the chapter, and the dashed arrows represent additional dependencies due to the presentation in this chapter. The dotted arrows represent a very weak dependency and lighter nodes are only present in the extended edition of this thesis:



### 2.1 Attribute Grammars with Side Effects

Ordered attribute grammars [Kastens, 1980] underly the extensions that we introduce in this chapter, and work out in the subsequent chapters of this thesis. In an OAG, attributes are evaluated in a fixed number of *visits* per node of the AST. Visits are a concept that play their role only in the evaluation algorithm of OAGs. In RulerCore, however, visits are a programming model: RulerCore has notation to specify visits so that the visits can be used to specify evaluation strategies.

## 2 Outline of the RulerCore Concepts

In Chapter 3 we introduce the concept of a visit and their notation. This concept plays a central role, because it provides a model of the evaluation of ordered attribute grammars. We show that this model is powerful enough to get the effect of a visitor-pattern based traversals as known from the OO-world in terms of an attribute grammar based description, and we show how to deal with monadic or side-effectful operations. In this section we explain what a visit is, and introduce the notation for specifying these visits.

**Visits and configurations.** We first explain what a visit is. For that, we consider the evaluation of attribute grammars. An attribute grammar describes correctly decorated trees, but not how such a decoration is to be constructed. For OAGs, there exists an evaluation algorithm that starts with an initially undecorated tree and finishes with a correctly decorated tree. In this process, we pass through a sequence of intermediate states, with each intermediate state corresponding to a partially decorated tree.

**Definition (State).** The *state* (or decorations) of a (partially) decorated tree consists of the local state of the root node of the tree and the states of its children. A local state of a node is a partial map from the attributes of the node to their values<sup>1</sup>.

**Definition (Defined attributes).** An attribute is *defined* when it is mentioned in the partial map.

Note that defined in this definition means that the attribute is part of the computed decorations of the tree. This definition is unrelated to rules defining attributes.

**Definition (Configuration).** A *configuration* is a set of inherited and synthesized attributes of the root of a (partially) decorated tree, and describes which attributes of the root have associated values in the (intermediate) state of the tree. There exists some total order  $\prec$  among configurations (we come back to this later). The total order  $\prec$  must be stronger than the subset relation among configurations.

Thus, a configuration is an abstract description of an intermediate state.

**Definition (Minimally defined state).** Given (static) dependencies between attributes, a tree is in a *minimally defined* state for a given configuration when precisely the attributes mentioned in the configuration have a value in the local state of the root and their (indirectly) dependent attributes have an associated value in the state of the tree.

**Definition (Visit).** A *visit* is a state and configuration transition, which takes a tree<sup>2</sup> in a state as described by a configuration  $A$  to a tree in a state as described by a different configuration  $B$  with  $A \prec B$ .

---

<sup>1</sup> How the value of an attribute is represented depends on the implementation or the host language. Such a value can be an element in the domain of the attribute, but may also be a thunk. We usually assume that the values of attributes are at least in weak-head normal form (Section 1.2.2).

<sup>2</sup> A tree does not have to be the full tree, but may also be a subtree. The definition is not limited to the full tree. Indeed, a visit may require visits to subtrees: we usually describe visits per node of the tree, and specify what visits are performed to children of the node.

Thus, during a visit, computations are performed which determine the values of attributes. Usually,  $B$  contains at least one synthesized attribute that is not yet present in  $A$ , since that is a motivation to perform a visit<sup>3</sup>. Values for the inherited attributes in the set difference  $B - A$  are provided by the parent<sup>4</sup> prior to the visit. The notation that we present below facilitates a statically finite and explicit description of state and configuration transitions induced by visits.

If we take a partial order among configurations instead of a total order, the order represents a Direct Acyclic Graph (DAG) where the vertices of the graph represent configurations and the edges represent visits. The evaluation of attributes for a tree associated to this DAG entails walking a path in the DAG where the parent chooses which visits to invoke on the tree. However, since we base our work on OAGs, we impose a total order that corresponds to a DAG with only a single path, which simplifies the implementation and the notation. In Section 2.6 we generalize our work so that the DAG is actually a tree.

**Rationale.** In this thesis, we distinguish two important notions of evaluation: a visit and a step. Visits (this section) provide a static model of evaluation, whereas steps (Section 2.5) provide a runtime model of evaluation. We argued in Section 1.5 that we wish to describe evaluation strategies, thus we do so in terms of the models as mentioned above. In this Section, we focus on the static model.

Every node in the tree is related to a production, and each production has an associated collection of rules. Attributes are computed by evaluating rules (Section 1.3.4). OAG evaluation is compositional in that it separates the evaluation of collections of rules of the parent from the evaluation of the children. In the case of evaluation of the parent, we make *internal visits* explicit, which statically describes the evaluation of a collections of rules of a node in the tree. In the case of the evaluation of children, we note that a visit to a child is the statically *smallest* unit of evaluation for a child that can be specified as part of the evaluation of the parent, and thus provides a model with a fine granularity.

**Description of visits.** RulerCore provides notation to describe visits. In a conventional AG, we declare attributes per nonterminal, and specify rules per production. In RulerCore, we additionally declare a linearly ordered sequence of visits and specify which attribute belongs to which visit. Also, we specify for each rule in which visit it is evaluated. Below, we describe the notation that forms an essential prerequisites for the remaining sections of this chapter. Chapter 3 provides extensive examples and technical background.

**Definition** (Interface declaration). An *interface* declaration of a nonterminal specifies a linear sequence of visits to the nodes with which the nonterminal is associated.

**Definition** (Visit declaration). Each visit declaration specifies which attributes must be defined prior to that visit, and which synthesized attributes become defined as a result of the

<sup>3</sup> In a conventional OAG, the *only* motivation for performing a visit is to get synthesized attributes computed, and this motivation is formalized as dependencies between attributes. Below, we show other motivations (e.g. to perform side effects) and show how these are formalized.

<sup>4</sup> We assume that each node has a parent. In case of the root node, the parent is represented by the interface with the host language (Section 1.3.1).

## 2 Outline of the RulerCore Concepts

visit.

Configurations are not explicitly declared. The configuration of a node before a given visit is the union of the attributes declared for preceding visits, starting with the empty set. Visit declarations thus form a partitioning of the attributes of a nonterminal.

In the following example, for some *Expr* nonterminal, we declare a linear sequence of visits *analyze* and *compile*. We specify that in the first visit *analyze* a synthesized attribute *errors* is defined given the inherited attribute *env*. In the second visit *compile*, the synthesized attribute *output* gets defined, given the inherited attribute *optimize*, and the attributes defined earlier:

```
itf Expr  visit analyze  inh env      :: Env
                               syn errors  :: Errs
      visit compile  inh optimize :: Bool
                               syn code    :: Code
```

The order of appearance of visits matters, whereas the order of appearance of attribute declarations in a visit-block does not.

As we have seen, configurations are an abstract representation of the state of a node. A configuration records which attributes have been evaluated. Configurations are not explicitly named in the interface declaration. Instead, we associate with a visit the configuration that corresponds to the state at the beginning of the visit. So, during evaluation, the decoration of an *Expr* node is initially in the *analyze* state, then in the *compile* state, and finally in some final state.

The above example declares two visits for nodes associated with the *Expr* nonterminal. During evaluation, these visits correspond to a state transition. Such a state transition is described by a collection of rules which are specified per production using a semantics-block in conventional AG notation (Section 1.3.1):

```
sem Expr prod Var                -- rules for production Var
  loc.defined = loc.x 'member' lhs.env -- tests whether ident loc.x is in the env
  lhs.code    = Code.Var loc.x      -- some translation to a target language
  lhs.errors  = if loc.defined then [] else [Undefined loc.x]
```

The above semantics-block introduces three rules for the production *Var*. Each rule is implicitly associated with a visit to *Expr*. Later we introduce notation to declare such a correspondence explicitly.

**Default-notation.** Before we continue with explicit notation for visits, we take a slight detour to introduce some notational conveniences. When programming with AGs, we often use copy rules (Section 1.3.12). Note that RulerCore is a *core language*, thus we prioritize implementation convenience over concise syntax. Copy rules are a typical front-end concept. However, copy rules may interact with RulerCore's evaluation algorithm, hence we model them explicitly.

RulerCore provides default-notation to improve on *copy rules*. For example, we can define the rules of production *App* with *default-rules*:

```

sem Expr prod App      -- rules of production App
  default env optimize   -- declares copy rules
  default errors = concat -- declares collection rule for errors

```

In the example, the default-rules introduce generic rules for the inherited attributes *env* and *optimize*, and a collection rule for the synthesized attribute *errors*.

A default-rule specifies that the value of an attribute can be inferred from equally named attributes of the production. We provide several flavors of default rules for different generic situations, which each differ in how attributes are combined. The above rules are syntactic sugar<sup>5</sup> for rules of the following form:

```

r ::= ...           -- conventional rules (Section 1.3.1)
    | m o x̄ h       -- default rules for the attrs with name x̄
m ::= default0    -- applies even if no attrs matched
    | default1    -- at least one attr must match (default)
o ::= lexical    -- uses the lexical order of children (default)
    | lexicalrev  -- uses the lexical order in reverse
    | scheduled  -- determines order after scheduling
h ::=  $\epsilon$     -- threaded behavior
    | = e         -- use expr e to combine a list of values of matching attrs
    | use e1 e2 -- applies the list algebra (e1, e2)
x

```

The order annotation *o* determines the order in which children are considered in the resolution process. In case of the scheduled-order annotation, the actual definition is determined after scheduling. As a notational convenience, a default-rule may be specified as part of the interface of a nonterminal, which then applies to all semantics-blocks of that nonterminal.

**Explicit association to visits.** In RulerCore, rules are associated with a visit. The association is by default implicit, but notation is available to specify the association explicitly by organizing the rules inside a visit-block:

```

sem Expr prod Var -- semantics-block for production Var
...
visit analyze      -- note the indentation (important in a later section)
...
visit compile     -- note the indentation (important in a later section)
...

```

With this notation, we specify constraints on the scheduling of rules, in addition to the conventional constraints imposed by value dependencies between attributes and rules. Via a cycle analysis, the constraints can be verified to be satisfiable using standard algorithms (Section 1.3.4). We come back to the rationale for the additional constraints later.

<sup>5</sup> We give the abstract syntax of the *desugared* rules, but use the sugared version in the code figures.

```

sem Expr prod Var                                -- rules of production Var
  visit analyze                                     -- rules of the analyze visit or later
    loc.defined = loc.x‘member‘ lhs.env
    lhs.errors  = if loc.defined then [] else [Err_Undefined loc.x]
  visit generate                                     -- rules of the generate visit
    lhs.code    = Code_Var loc.x                    -- actually independent of any visit
sem Expr prod App                                  -- rules of production App
  default env optimize                              -- declares copy rules
  f.env     = lhs.env                                -- explicitly written rule
  visit generate                                     -- rules of the generate visit
    a.optimize = lhs.optimize                       -- explicitly written rule
    lhs.errors  = concat [f.errors,a.errors]        -- explicitly written rule

```

**Figure 2.1:** Examples of organizing rules in a visit-block.

**Definition** (Visit semantics). A visit-block  $t$  is explicitly associated to some visit  $x$  and may contain rules and a nested visit-block. The rules may be evaluated during visit  $x$  or a later visit.

The following is the grammar of a semantics and visit-block:

```

s ::= sem N prod P  $\bar{r}$  t    -- common rules  $r$  and visit blocks  $t$ 
t ::= visit x  $\bar{r}$  t          -- common rules  $r$  and subsequent visit  $t$ 
    |  $\epsilon$                 -- terminator of sequence of visit blocks (implicit)

```

A visit-block is associated with a similarly named visit declared on the interface of the nonterminal. However, not every visit is necessarily associated with a visit-block. The same name may not occur twice, and the total order must be preserved: If  $x$  is the name of a visit-block and  $x'$  the name of a nested visit-block then  $x \prec x'$ .

Figure 2.1 gives an alternative way to organize the rules of the earlier example. The rules inside a visit-block may appear in any order without affecting the semantics of the grammar.

A visit-block introduces a scope for local attributes such as *loc.defined*. When defined in a visit, such an attribute is only visible inside the visit-block it is defined in, and its enclosed visit-block. The scoping plays a role in Chapter 5 where rules may additionally be organized in clauses. The inherited and synthesized attributes of the children and *lhs* are globally scoped per production.

**Exploiting visits: invoke rules.** The additional refinements on the scheduling of rules can be used to allow side-effects in our specification (motivated later). For this purpose, we introduce two additional forms of rules.

**Definition** (Invoke rule). An invoke rule specifies properties of a visit to a child. These properties usually specify some evaluation strategy.

The following is the grammar of invoke-rules of which we explain some properties below:

```

 $r ::= \dots$ 
  | invoke  $x$  of  $\bar{c}$   $z$  -- specifies visit  $x$  of children  $\bar{c}$  with strategy  $z$ 
 $z ::=$  implicit -- determined only by attribute dependencies
  | explicit -- invoke-rule restricted to the visit it appears in
  | parent -- invoke-rule restricted to visit with the same name

```

The strategies are explained later. Invoke-rules are optional. When a visit  $x$  to a child  $c$  is not explicitly specified, it is implicitly specified as the rule:

### **invoke** $x$ of $c$ **implicit**

The invoke-rule is annotated with a strategy, which provide a means to specify properties of the evaluation of a child. The above strategies constrain when visits to children can be performed. We will later see more strategies.

The implicit-strategy (above) specifies no additional constraints. The explicit-strategy (above) requires the invoke-rule to be nested in a visit-block, and constrains the visit to the child to the evaluation of that visit-block. The parent-strategy requires that the parent has an equally named visit and constrains the visit to the child to that visit of the parent. An empty list of children in the invoke-rule applies the strategy to all children which have a visit defined with the same name.

**Exploiting visits: rules with side effects.** We allow rules with side effects, but restrict these to the introduction of children only.

**Definition** (Side-effectful child-rule). A *side-effectful child-rule* is a child defined by some impure expression.

A child-rule must appear in a visit-block, and its application is restricted to that visit. The syntax of child-rules is (similar to Section 1.3.7):

```

 $r ::= \dots$ 
  | child  $c : N \leftarrow f [\bar{a}]$  -- definition of child with side effects

```

The scheduling guarantee is that the impure expression  $f$  is applied before the end of the visit. The relative order of the side effects within a visit is however not specified<sup>6</sup>. This coarse-grained way of specifying the evaluation order allows us to safely integrate side-effectful operations in the attribute grammar, while not having to micro-manage the order of the side effects.

In case of Haskell as target language,  $f$  is a monadic expression that yields the structure of child  $c$  given values of attributes  $\bar{a}$ . For example, along the lines of Section 1.3.12, we can define such children to encode operations that provide fresh type variables and perform unification. Instead of threading a substitution, we pass these children an inherited *IORef* to

<sup>6</sup> We actually provide a notion of *internal visits* to specify the relative order of side effect within a visit.

## 2 Outline of the RulerCore Concepts

```
sem_App s_f s_a = lhs_analyze where           -- body of the function
  lhs_analyze lhs_env = do                   -- body of the first visit
    f_analyze ← return s_f                   -- monadic child rule
    a_analyze ← return s_a                   -- monadic child rule
    let f_env = lhs_env
        a_env = lhs_env
        (f_errs, f_compile) ← f_analyze f_env -- invoke rule
        (a_errs, a_compile) ← a_analyze a_env -- invoke rule
    let lhs_errs = concat [f_errs, a_errs]
        lhs_compile lhs_optimize = do ...   -- body of the second visit
    return (lhs_errs, lhs_compile)           -- results of first visit
```

**Figure 2.2:** A sketch of the coroutine *lhs\_analyze*.

a substitution. This allows us to schedule the effects of unification in a less strict way than the explicit threading of substitutions would entail<sup>7</sup>.

**Scheduling and coroutines.** In the context of this chapter, we shall refer to augmented production dependency graphs as PDGs. Section 1.3.2 explains how dependency graphs are obtained from AG descriptions. In this section, I/O graphs are Nonterminal Dependency Graphs (NDGs) because of the explicitly declared single visit sequence per nonterminal. From the PDGs, an execution plan with an as-late-as-possible scheduling of the rules can be obtained if the PDGs are cycle-free. Since we introduced new rules, the question arises how these rules affect the evaluation of the grammar. Also, RulerCore’s NDGs leave less freedom for scheduling the rules in the PDGs, since we need to adhere to the explicitly defined interfaces.

We introduce the language RulerBack as a desugared variant of RulerCore which represents execution plans<sup>8</sup>. In comparison to RulerCore, in a RulerBack all implicit syntax is made explicit and rules in a RulerBack description are totally ordered. In this thesis, we define mappings of RulerBack to algorithms in various host languages.

Section 1.3.5 shows that ordered AGs can be implemented with coroutines. The example below serves as a sketch of a mapping from RulerBack to *monadic* coroutines. For each production  $P$ , we introduce a semantic function  $sem_P$ , which takes the coroutines  $s_f$  and  $s_a$  as parameter that serve as children  $f$  and  $a$  and produces a coroutine  $lhs\_analyze$  for the first visit  $analyze$ . The coroutine for some visit  $x$  is a function  $lhs\_x$  that takes the values of the inherited attributes of visit  $x$ , and produces a monadic tuple with values of the synthesized attributes of visit  $x$  and a coroutine for the successor of  $x$ . The coroutine  $lhs\_optimize$  (Fig-

<sup>7</sup> In Chapter 5 we show how to treat nodes as first class value, and show in Middelkoop [2011b] some complex traversal patterns that do not follow the tree structure. In such situations, we need more flexibility in the scheduling of side-effectful operations.

<sup>8</sup> RulerCore can be considered a programming language for execution plans of AGs.



ure 2.2) is thus constructed as part of the body of *lhs\_analyze*. An attribute  $k.x$  is transcoded as the Haskell identifier  $k\_x$ . The monad serves as an abstraction for evaluation algorithms. We exploit the monadic structure in later chapters.

**Foundation.** The fragment of RulerCore that we introduced so far does not add to the expressive power of attribute grammars: it can be expressed as a conventional attribute grammar, which we do so below to be more precise about the semantics of the notation. We assume that the RulerCore description is desugared to RulerBack, which we mentioned earlier. Below, we describe how to map the RulerBack description to a conventional attribute grammar. In general, interface declarations are mapped to attribute declarations by erasing visits. Semantics-blocks in RulerCore are translated to semantics-blocks by erasing visit-blocks and invoke-rules.

To represent the erased information in a conventional attribute grammar, we thread an additional attribute through the tree for each visit. These attributes serve as synchronization points for the beginning and end of the visit. Per visit  $v$  of a nonterminal  $N$ , we introduce two additional attributes  $begin_v$  and  $end_v$ :

$$\begin{array}{ll} \mathbf{attr} N & \mathbf{inh} \begin{array}{l} begin_v \end{array} :: S T \\ & \mathbf{syn} \begin{array}{l} begin_v \end{array} :: S T \end{array}$$

The attributes model the side effects by encapsulating a state as some type  $S T$ , which we come back to later.

Also, we thread these attributes *through the rules* to enforce their evaluation order. Per production, we associate a unique consecutive number (starting from 0) with each rule in a production. This is possible because there exists a total order among the rules. For each rule, we introduce a local attributes  $loc.begin_i$  and  $loc.end_i$  where  $i$  is the number associated with the rule. The purpose of the attributes is to mark respectively the beginning and end of the evaluation of the rule. We show later how to make a rule dependent on its *begin* attribute (in addition to its normal dependencies), and how to make the *end* attribute dependent on the evaluation of the rule.

For a visit  $v$  and production  $P$  of nonterminal  $N$  there exists a collection  $R$  of rules that are associated with  $v$  after scheduling. If this collection is not empty, let  $k$  be the lowest number associated with the rules, and  $l$  the highest number. We then connect the begin and end of the visit with the beginning and end of the rules associated with the visit:

$$\begin{array}{ll} \mathbf{sem} N & \mathbf{prod} P \\ & lhs.begin_v = lhs.end_j \quad \text{-- if } R \text{ is empty} \\ & loc.begin_k = lhs.begin_v \quad \text{-- otherwise } R \text{ not empty} \\ & lhs.end_v = loc.end_l \quad \text{-- otherwise } R \text{ not empty} \end{array}$$

When a rule with associated number  $i$  and a rule with number  $i + 1$  are in  $R$ , then we add also:

$$\begin{array}{ll} \mathbf{sem} N & \mathbf{prod} P \\ & loc.begin_{i+1} = loc.end_i \end{array}$$

## 2 Outline of the RulerCore Concepts

```

class ThrEff t where
  type M t :: * → * -- type of a monadic computation
  type S t :: *      -- type of the state
  impure :: M t  $\alpha$  → S t → (S t,  $\alpha$ )
  pure   ::  $\alpha$       → S t → (S t,  $\alpha$ )
  pure (x, s) = s 'seq' x 'seq' (s, x)

```

**Figure 2.3:** API of threaded effects.

```

data IsPure -- do not thread a state
instance ThrEff IsPure where
  type M IsPure t = t
  type S IsPure   = ()
  impure          = pure
data IsIO -- thread state of the world
instance ThrEff IsIO where
  type M IsIO     = IO
  type S IsIO     = State# RealWorld
  impure (IO m) w = case m w of (# w', a #) → a 'seq' (w', a)

```

**Figure 2.4:** Example instances for threaded effects.

At this point, we chained the attributes together, except that still attributes  $loc.begin_i$  needs to be connected to  $loc.end_i$ , and the begin and end of visits to the children need to be incorporated.

To thread the attribute through a rule, we introduce in Figure 2.3 the functions  $pure$  and  $impure$  which depend on the type  $T$ . The function  $impure$  takes an effectful computation  $M t \alpha$  and an initial state  $S t$ , then produces an updated state paired with the result of the computation. The function  $pure$  passes the state on unchanged. Figure 2.4 gives some exemplary instances. An instance of  $ThrEff$  can be given for any monad that threads a state.

To complete the chain, we show the mapping of rules of RulerCore. The above functions are used in the translation of a rule  $r$  with associated number  $i$  to  $\llbracket r \rrbracket_i$ :

$$\begin{aligned}
 \llbracket \text{invoke } v \text{ of } c \rrbracket_i &\rightsquigarrow c.begin_v &&= loc.begin_i \\
 & &&loc.end_i &&= loc.end_v \\
 \llbracket \text{child } c:N \leftarrow f [\bar{a}] \rrbracket_i &\rightsquigarrow (loc.end_i, loc.c) &&= impure (f [\bar{a}]) loc.begin_i \\
 & &&\text{child } c:N &&= loc.c \\
 \llbracket p [\bar{a}_2] = f [\bar{a}_1] \rrbracket_i &\rightsquigarrow (loc.end_i, p [\bar{a}_2]) &&= pure (f [\bar{a}_1]) loc.begin_i
 \end{aligned}$$

The concept of visits thus provides a means to reason about attribute grammars with side-

effective computations in their rules. Note that although the AG description may be thought of as having side effects, the underlying model is still purely functional.

**Remarks.** We purposefully allow side effects only in the creation of children. Conventional rules must be purely functional. This ensures that the attributes have a referentially transparent definition, even though the tree structure itself not<sup>9</sup>. Chapter 6 shows how to implement search algorithms, and introduces syntax to define children without inherited attributes. For advanced search algorithms, which make use of sharing and memoization, limited use of side effects plays an important role.

More generally, by making visits explicit, we can integrate our approach with compilers that are built on top of monads, and to use tree traversals in impure environments. Also, we can use the side effects to efficiently access results from nodes visited earlier in a traversal. This can be used to implement memoization strategies.

The visitor pattern [Gamma et al., 1993] is often employed to implement recursive traversals over tree structures in imperative languages. Concretely, Chapter 3 presents how our approach generalizes over the visitor design pattern. For this purpose, we use JavaScript as a host language, which in passing shows that our extensions are applicable to domains other than the implementation of type inference algorithms or functional programming languages. In this context, a *visitor* is an object that contains an algorithm that describes which children to visit, and what changes to apply to the state of the node, or the visitor itself. Attributes provide a convenient way to access the state of nodes through attributes, and with our approach the changes to the state of the visitor can be encoded with side effects. With respect to these attributes, our approach offers the benefits of AGs, including the static enforcement that attributes are defined before they are used.

## 2.2 Attribute Grammars with Commuting Rules

In Chapter 4, we generalize visits to a *phases*. A visit is a technical more internal concept which precisely controls the evaluation of the grammar. A phase is a more abstract concept which the programmer can use to specify properties of the evaluation of the grammar. To reason with side effects in this setting, we present *commuting* rules, which are rules with relaxed dependencies.

**Phases.** We start with the notion of a phase:

**Definition (Phase).** A *phase* represents a sequence of state transitions, controlled and observable by the parent, which take the node's state to a state described by its next configuration.

Such a state transition consists of a sequence of smaller state transitions, which correspond to the visits as described in Chapter 3. A nonterminal may be associated with a set of possible visit sequences for its phases, and a production specifies for each of its children which sequence to take.

---

<sup>9</sup> We can encode any rule as an attribute-defined child (Section 1.3.7), thus the restriction does not limit expressiveness. The purpose of the restriction is to ensure that side effects are sufficiently contained.

## 2 Outline of the RulerCore Concepts

```
itf Block -- phase interface (visit-interfaces are not given anymore)
phase analyze -- analyze phase
  inh pred :: Lab -- label to be used as predecessor from the predecessor
  inh succ :: Lab -- label to be used as successor from the successor
  syn pred :: Lab -- label to be used as predecessor for the successor
  syn succ :: Lab -- label to be used as successor for the predecessor
phase transform -- transformation phase
  inh debug :: Bool
  syn trans :: SELF -- self attribute (Section 1.3.6)
default debug -- default rule on interface for inh attr
```

**Figure 2.5:** Phase interface of a *Block*.

We illustrate the above with an example. Suppose that we describe an analysis and transformation of a tree of labeled instruction blocks. The abstract syntax of blocks is described by the following grammar:

```
grammar Block prod Seq nonterm l, r :: Block
                prod Leaf term lab :: Lab
                term instr :: Instr
```

The actual transformation of the instructions is out of the scope of this example. Let *transform* be a function that requires the label of the predecessor and successor to transform the instructions in the leaves.

To apply the *transform* function, we associate the label of a preceding and succeeding block with each instruction. The chained attribute *pred* represents the label of the left-nearest instruction, and attribute *succ* the label of the right-nearest instruction. Effectively, we pass *pred* from left to right, and *succ* from right to left. The *phase interface* for a nonterminal declares these attributes is given in Figure 2.5. Attributes may be declared outside phases. The ordering of phases is deduced from semantics blocks, thus not from the order of appearance in the phase-interface specification: phases represent non-overlapping units of evaluation.

As part of the semantics for productions of *Block*, we describe the flow of the *pred* and *succ* attributes in Figure 2.6. Seq-productions act as crossbar switches, and Leaf-productions inject their labels in the attribute flows. Since we have inherited and synthesized attributes with the same name, we use the prefixes *inh* and *syn* to explicitly distinguish these attributes. Assume that *Block* is also the root symbol, for which we at the root provide initial values for the inherited attributes, and request values for all synthesized attributes.

**Implementation of phases.** Since a phase effectively represents a unit of evaluation, we can choose an algorithm for the evaluation of a phase. We assume here that we choose a statically ordered evaluation algorithm, which reduces the choice to either a Kastens style or a Kennedy-Warren style algorithm (Section 1.3.5).

```

sem Block prod Seq  -- rules related to pred and succ
  inh.l.pred  = inh.lhs.pred  -- left to right
  inh.r.pred  = syn.l.pred
  syn.lhs.pred = syn.r.pred
  inh.r.succ  = inh.lhs.succ  -- right to left
  inh.l.succ  = syn.r.succ
  syn.lhs.succ = syn.l.succ

sem Block prod Leaf
  syn.lhs.pred = loc.lab      -- is predecessor of next
  syn.lhs.succ = loc.lab      -- is successor of prev
  loc.newInstr = transform loc.instr inh.lhs.pred inh.lhs.succ lhs.debug
  syn.lhs.trans = Leaf loc.lab loc.newInstr

```

**Figure 2.6:** The semantics of productions of *Block*.

A Kastens-style algorithm does not suite the example. In the example, the rules for *pred* and *succ* are independent. However, the attributes of the analyze-phase need to be computed in at least two visits. The rules for production *Seq* require either *succ* or *pred* to be computed first. This is a typical example where a Kastens-style scheduling [Kastens, 1980] fails to find an ordering, because that scheduling induces extra edges in the PDG, which for this example causes a cycle.

Kennedy and Warren [1976] describe an algorithm that does not induce extra edges in the PDG. A set of visit sequences is determined for each nonterminal, such that there is one visit sequence per context of an occurrence of the nonterminal symbol. We present a variation on this algorithm that schedules rules as late as possible, and only those that needed in a given context. Moreover, we show how to represent such visit sequences in a strongly-typed functional language.

**Visits-DAG.** We associate a graph structure with each nonterminal which represents the visits of that nonterminal:

**Definition** (Visits-DAG). A *visit-interface DAG* describes the set of visit interfaces that are associated with a nonterminal. The graph has exactly one source vertex, and at least one sink vertex. Each vertex represents a configuration, each arrow a visit, and each path from the source to some vertex a visit-interface. To disambiguate, we may call a vertex in the visits-DAG a *visits-vertex*.

For the above example, Figure 2.7 shows the visits-DAG. There are at least three paths in the visits-DAG. The middle path represents the root where all inherited attributes are available, and two other paths represent the respective first knowledge of one of the inherited attributes.

## 2 Outline of the RulerCore Concepts

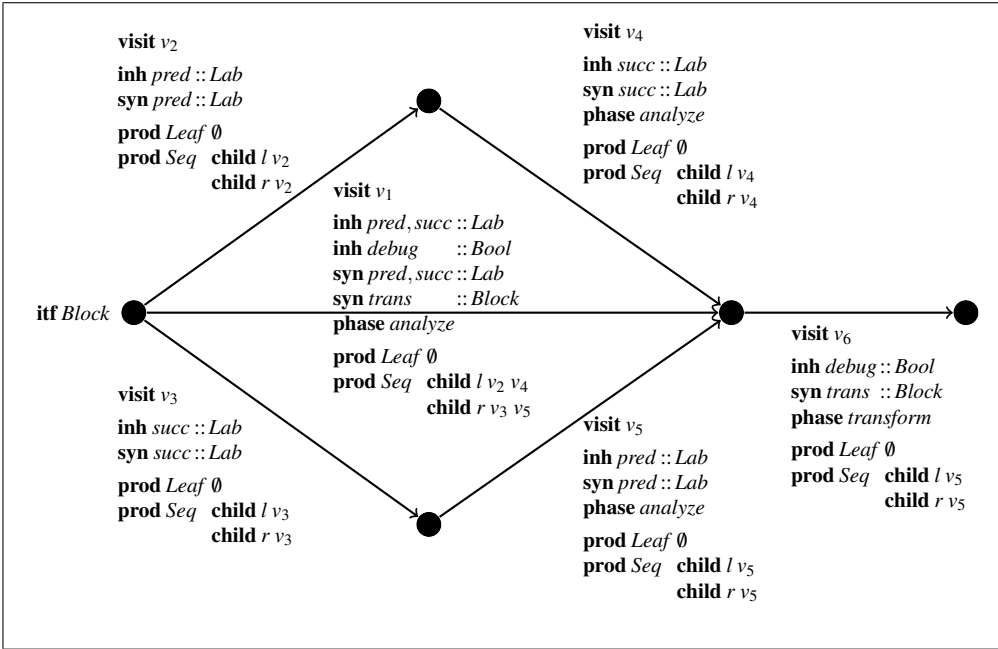


Figure 2.7: Visits-DAG of the example.

Each edge has at least one *output*, which is either a synthesized attribute or phase ending. Along each path, the number of attributes increases. Each edge corresponds with a visit; we gave each a unique label  $v_i$ . Also, we associated with each edge some meta-data regarding productions: the visits performed on the children of the productions during the execution of visit that is associated with the edge.

Paths may be of different length, and end in different configurations. In the above example, all paths end in the same configuration because in each context all attributes are eventually needed.

Section 4.5 describes the visits graph and its properties in more detail and shows how to incrementally compute it. Given this graph, for each edge and each production, a collection of RulerBack rules can be determined. The branching-factor of each node determines code duplication. In practice, this code duplication is not a reason for concern. The visits graph of the largest AG of UHC has about 10,000 edges and already leads to a tractable implementation. With some optimizations (Section 4.6), we reduced this number to about 3,000 edges.

**Commuting rules.** Section 2.1 shows a translation of visits to conventional attribute grammars using a functional encoding of the state. The translation involved adding an additional chained attribute (the state attribute) per visit which represents the state and a transformation of the rules to thread the attribute through the rules scheduled for the visit. A property of this approach is that the side effects that arise from visit a child of a parent can only be observed

by the parent or its other children by inspecting the state attribute of the child after the visit.

A similar translation is possible for phases. Analogously, a parent only observes the side effects arising from a child after completing the phase. However, during the evaluation of a phase of a child of a parent, side-effectful rules of the parent or the other children of the parent may be evaluated, since a phase consists possibly of multiple visits to a child. In this situation, with a single chained attribute per phase, side effects arising from a child may not be timely observed in the parent or in siblings, and such a translation does not fully capture the semantics of side-effectful rules.

A possible approach is to translate the phases to an explicit visit sequence, and then use the translation of Section 2.1. However, visits are implicit in the phases model and additionally there may be a visit sequence per context. Instead, we take the opportunity to present *commuting rules*.

**Definition** (Commutative compositions and commutable rules). Given an explicit ordering of rules, the composition of two rules is *commutative* when the two rules are *commutable*, which means that the rules may be swapped in the composition without affecting the intended result.

Rule composition is a conditionally commutative operator. Commutable rules can be considered as commutable operations. The swapping of rules models side effects, and commutativity permits reasoning about the safe use of side effects.

A semantic tree is a composition of the rules of the tree (Section 1.3 and Section 1.3.4). Section 1.3.9 shows that a composition of rules can be expressed with arrow notation, which is a convenient notation to define when two rules are commutable. Further, we wish to refer describe a composition of rules (e.g. the composition of rules of a node) in a larger context (e.g. the composition of rules in a tree). We define a *rule context*  $h$  as a composition of rules with a hole in it so that  $h c$  represents the composition of rules with  $c$  the composition of rules at the location of the hole.

**Definition** (Commuting rule). A *commuting rule* is a rule of the form  $(x', y') = f x y, x \diamond x'$  where the letters  $x, y$ , etc. are (sets of) attribute occurrences and  $f$  is a semantic function.

A commuting rule thus only differs in the notation  $x \diamond x'$ , which denotes that the rule may be swapped with rules that define  $x$  or use  $x'$  (with renaming of the attribute occurrences). Such a rule is said to commute over  $x$  and  $x'$ .

Consider a rule  $r_1$  of the form  $(x_1, y_1) = f(x_0, y_0), x_0 \diamond x_1$  and a rule  $r_2$  of the form  $(x_2, y_2) = g(x_1, y_1), x_1 \diamond x_2$ . In a composition of rules containing  $r_1$  and  $r_2$ , the additional notation specifies that  $r_1$  may appear ordered before  $r_2$  and vice versa. Without the additional notation the rule  $r_1$  must appear before  $r_2$  because  $r_2$  refers to an attribute defined by  $r_1$ .

With commuting rules as AG feature, side effects can be encoded as a single chained attribute per nonterminal threaded through each rule and having each rule commute over this attribute. Such a translation is more straightforward than the translation in Section 2.1 and also works for phases.

**Referential transparency.** At the level of specification, the use of commutable rules may break referential transparency and thus complicate equality reasoning. A question that arises

## 2 Outline of the RulerCore Concepts

is how to reason with a safe use of commutable rules. We define below a law for this purpose.

Suppose that  $r_1 \not\prec r_2$  denotes that  $r_1$  is independent of  $r_2$  with respect to the dependencies between attributes and rules except for the dependencies between attributes where the rules commute over. There are two compositions in arrow notation ( $c_1$  and  $c_2$ ) to consider:

$$\begin{aligned}
 c_1 &= \mathbf{proc} (x_0, y_0, z_0) \rightarrow \mathbf{do} && \text{-- composition of } r_1 \text{ and } r_2 \\
 &\quad (x_1, y_1) \leftarrow f \prec (x_0, y_0) && \text{-- rule } r_1 \text{ in arrow notation} \\
 &\quad (x_2, z_1) \leftarrow g \prec (x_1, z_0) && \text{-- rule } r_2 \text{ in arrow notation} \\
 &\quad \mathbf{returnA} (x_2, y_1, z_1) \\
 c_2 &= \mathbf{proc} (x_0, y_0, z_0) \rightarrow \mathbf{do} && \text{-- composition of } r_2 \text{ and } r_1 \\
 &\quad (x_1, z_1) \leftarrow g \prec (x_0, z_0) && \text{-- rule } r_2 \text{ in arrow notation} \\
 &\quad (x_2, y_1) \leftarrow f \prec (x_1, y_0) && \text{-- rule } r_1 \text{ in arrow notation} \\
 &\quad \mathbf{returnA} (x_2, y_1, z_1)
 \end{aligned}$$

The identifiers  $y_0$  and  $z_0$  represent the independent input attributes of respectively  $r_1$  and  $r_2$ , and identifiers  $y_1$  and  $z_1$  their respective independent output attributes. Identifiers  $x_0$ ,  $x_1$  and  $x_2$  represent the attributes over which the rules may commute.

**Definition.** Rule context A rule context  $h$  is a function that serves as an abstraction of a rule composition with a hole. It takes as parameter the composition to fill to hole with.

**Definition** (Commutable over attributes). We now say that  $r_1$  and  $r_2$  are *commutable* over attributes of  $x_0, x_1$  and  $x_1, x_2$  if  $r_1 \not\prec r_2$  if their compositions  $c_1$  and  $c_2$  give an equivalent results  $h c_1 = h c_2$  in some given rule context  $h$ .

When the rules are commutable, the outcome of swapping the rules in rule context  $h$  is equivalent, thus the slide of the grammar that contains the rules  $r_1$ ,  $r_2$  and those in  $h$  is referentially transparent. Rule context  $h$  should be chosen in such a way that it expresses how the context of the rules interprets the attributes computed by the rules.

We finish this discussion of commutable rules below with an exemplary definition of  $h$  which states that for an attribute that provides fresh numbers only uniqueness is relevant.

For some exemplary grammar of expressions, the following two rules thread a counter  $k$ , and extract two unique numbers in  $loc.u_1$  and  $loc.u_2$ :

$$\begin{aligned}
 \mathbf{sem} \text{ Expr } \mathbf{prod} \text{ Var} \\
 (loc.k, loc.u_1) &= f (inh.lhs.k, inh.lhs.k), \quad inh.lhs.k \diamond loc.k && \text{-- rule one} \\
 (syn.lhs.k, loc.u_2) &= g (loc.k + 1, loc.k), \quad loc.k \diamond syn.lhs.k && \text{-- rule two}
 \end{aligned}$$

With the following definitions for  $f$  and  $g$ :

$$\begin{aligned}
 f (a, -) &= (a + 1, a) \\
 g (b, -) &= (b + 1, b)
 \end{aligned}$$

For the following definition of  $h$ , the above two rules are commutable. In both compositions of the rules, the two resulting numbers are different from each other:

$$\begin{aligned}
 h r n = a \neq b \mathbf{where} &&& \text{-- abstraction: the unique numbers should be distinct} \\
 (-, a, b) = r (n, (), ()) &&& \text{-- for any number } n \text{ that represents the } inh.lhs.g
 \end{aligned}$$



We may choose functions  $h$  that state stronger invariants and take more context into account. For example, when we consider the collection of a list of error messages, we may take the slice of the rules that depend on the error messages, and require that the lists are equal when ordered according to the source location of each message.

Commutable rules can be applied when expressing collection attributes as a chained attribute. A collection attribute is a synthesized attribute with a commutative monoid or trace monoid as value. Often, such attributes can be encoded more efficiently as a chained attribute. However, threading a chained attribute through some children may induce tighter dependencies than combining synthesized attributes of these children, and thus reduces freedom in attribute scheduling and may even lead to cycles. With commutable rules a chained attribute can be used without the tighter dependencies.

**Remarks.** In Chapter 4 we work out phases in more detail. In this chapter, we describe how to compute the dependency graphs and how to perform scheduling of phases. This work shows how to generalize visits to phases, and allows us to describe the extensions that we present in the next sections (and their corresponding chapters) using visits, so that we factor out the dependency graphs and scheduling in the next sections and chapters.

## 2.3 AGs with Tree Construction

In Chapter 5, we show several AG extensions based on the model of explicit visits that allow us to conditionally and iteratively define attributes and children. Additionally, we use annotations on visits and invocations of visits to fine-tune evaluation strategies.

In this section, we give an overview of the extensions. In a conventional attribute grammar the rules to evaluate for a node are the rules associated to the production that is associated to that node. As extension, we wish to have more fine-grained control over which sequence of rules is evaluated. Therefore, we split up visit-blocks in a sequence of clause blocks. Each clause-block may contain rules, and per visit some strategy chooses which clause-block to use to compute the attributes. In this section, we take a fixed strategy based on the order of appearance of clause-blocks and backtracking. We show that with this approach we can implement a dispatch of rules based on values of inherited attributes. In Section 2.5 we present a mechanism based on a stepwise evaluation to actually define custom strategies.

Further, we show how to express iteration by annotating invoke-rules with a strategy that repeats the evaluation of the visit until a condition is met. Moreover, first-class children are an extension that allow children to be detached as value, or attached from a value. The techniques combined provide a powerful mechanism to encode fixpoint computations.

In the remainder of this section, we explain these extensions one-by-one. In Chapter 5 we show how these extensions are implemented.

**Clauses.** Instead of associating a collection of rules per production, we organize the rules in a different way. We associate a DAG with a production. Each vertex is associated with a configuration and each edge is associated with a visit and with a collection of rules. There may be multiple vertices associated to the same configuration, although there may only be

## 2 Outline of the RulerCore Concepts

```

itf Expr visit check inh env  :: Env  -- environment containing declared types
      inh tp      :: Ty    -- expected type of the expr
      syn errors :: Errs  -- result of type checking

sem Expr prod Var visit check  -- cases for the check-visit of the var-prod
clause defIdent                -- case for when the variable is in the env
  match (Just loc.declTp) = lookup loc.nm lhs.env
internal matchTp              -- internal case distinction
  clause typeOk                 -- case for when the type matches
    lhs.errors = []
    match True = lhs.tp 'isInstance' loc.declTp
  clause typeFail               -- case in case of a type mismatch
    lhs.errors = [Mismatch lhs.tp loc.declTp]
clause undefIdent             -- case for when the variable is undefined
    lhs.errors = [UndefVar loc.nm] -- assumes a match of defIdent failed

```

**Figure 2.8:** Example of clauses.

one source vertex, which must be associated with the empty configuration. During evaluation, a path is traversed through the DAG: one edge per visit and a strategy associated with vertices dictate which edge to traverse.

**Definition** (Clause). A *clause* is an edge in the DAG as specified above.

As notational simplification, we impose the restriction that the DAG<sup>10</sup> must be a tree, and present notation below on how to describe this tree. Essentially, the rules are organized in clause-blocks per visit-block.

We start with an example in Figure 2.8 before explaining the notation. The example consists of a type checker for some var-production of a lambda calculus. In the example, we use clauses to encode case distinction. We distinguish a clause *defIdent* for when the identifier is in the environment, and a clause *undefIdent* when this is not the case. Moreover, we split the *defIdent* clause in two more clauses depending on whether the expected type matches the declared type using an internal visit.

The nesting of clauses forms a decision tree. A path in this tree is the sequence of clauses that are selected to compute the outputs of the visit. For now, we assume that clauses are selected with a fixed strategy based on the order of appearance (to which we come back later). Internal visits can be considered as  $\varepsilon$ -edges in the DAG as mentioned above.

The following changes to notation allow visits to consist of a non-empty ordered sequence of clauses:

<sup>10</sup> We thus identify two important DAGs: a DAG per nonterminal which describes visits and attributes, and a DAG per production which describes different sets of rules for visits to compute the attributes. The restrictions that we impose on the DAGs simplify the implementation or the notation.

```

 $t ::= \mathbf{visit} \ x \ \bar{r} \ \bar{k}$  -- conventional visit block, common rules  $r$ , and alternatives  $\bar{k}$ 
    |  $\mathbf{internal} \ x \ \bar{r} \ \bar{k}$  -- internal visit block, common rules  $r$ , and alternatives  $\bar{k}$ 
    |  $\epsilon$  -- terminator of visit/clause branch (optional)
 $k ::= \mathbf{clause} \ x \ \bar{r} \ t$  -- clause-block with rules  $\bar{r}$  and visit  $t$ 
 $x$  -- identifier of a visit or clause

```

Each clause contains a set of rules. This set of rules defines the synthesized attributes of the visit, and potentially subsequent visits. Thus, each clause provides a number of alternative definitions of the synthesized attributes of a visit.

We distinguish conventional visits and *internal visits*. A conventional visit is invoked by the parent and declared as part of the interface of the nonterminal. Internal visits and clauses are evaluated as part of the evaluation of their encapsulating visit or clause.

To describe the clause selection strategy, we distinguish three types of outcome for rules, visits, and clauses. Evaluation either *succeeds* with resulting attribute bindings, *terminates exceptionally*, or *fails* with a recoverable failure:

- Clauses are evaluated in the order of their appearance. The first clause that succeeds or terminates is chosen as the clause that provides the outcome of the visit. If a clause fails, the next clause is evaluated.
- During the evaluation of a clause, the rules are evaluated in a scheduled order. When a rule succeeds, the next rule is evaluated. If all rules succeed, the clause succeeds. However, a clause fails if the evaluation of a rule terminates exceptionally or fails.
- A visit terminates if any of its evaluated clauses terminate, and otherwise succeeds if a clause succeeds. If all of its clause fail then the visit terminates exceptionally or fails recoverable. The respective difference is made by whether the visit is annotated as *total* or annotated with a *partial* strategy. Visits are declared as total by default.

There are two types of rules that may fail:

- An invoke-rule may be annotated with a *partial* strategy. If it is, the invoke-rule fails if the visit to the child fails. Otherwise the invoke-rule either succeeds or terminates exceptionally.
- We present *match-rules* to specify conditions. A rule **match**  $p = e$  requires that the value of  $e$  satisfies the pattern  $p$ , otherwise evaluation for the match-rule fails.

In the rule ordering, match-rules must be evaluated as part of the clause in which it is declared. Moreover, match-rules take priority in the rule scheduling. If two match rules are independent of each other, then the order of appearance determines which rule is scheduled first. As notational convention, we usually write match-rules up front in code examples.

Chapter 3 describes the evaluation of clauses as a generalization of productions. Chapter 5 describes an implementation in Haskell. Also, Chapter 6 shows how to evaluate clauses simultaneously.

In comparison to Conditional Attribute Grammars [Boyland, 1996] or conditionally defined rules in general, clauses allow us to define a condition for multiple attributes and also children.

```

grammar Eq prod   Check           -- multiple clauses below
itf Eq visit check -- checks type equality
  inh tp1  :: Type
  inh tp2  :: Type
  syn errs  :: Errs                -- outcome of check
sem Eq prod Check visit check -- semantics of Check prod
  default errs = concat           -- collect type errors
  clause twoInts                  -- when tp1 and tp2 are Ints
    match Ty_Int                  = lhs.tp1
    match Ty_Int                  = lhs.tp2
  clause twoArRs                  -- tp1 and tp2 both arrow-types
    match (Ty_Arr loc.a loc.b) = lhs.tp1 -- tests if lhs.tp1 is an arr
    match (Ty_Arr loc.c loc.d) = lhs.tp2 -- tests if lhs.tp2 is an arr
    child u1 : Eq                = sem_Check -- recursion on both arg-types
    child u2 : Eq                = sem_Check -- recursion on both res-types
    u1.tp1 = loc.a;  u1.tp2 = loc.c -- definitions of inh attrs of u1
    u2.tp1 = loc.b;  u2.tp2 = loc.d -- definitions of inh attrs of u2
  clause mismatch                 -- catch-all clause
    lhs.errs = [Err_Mismatch lhs.tp1 lhs.tp2] -- error for each mismatch

```

**Figure 2.9:** Matching example.

**Multi-attribute dispatch.** Clauses provide a convenient way to describe the structure of a derivation tree when the structure of the tree depends on the values of attributes. For example, to prove type equality, the structure of the derivation tree is determined by two attributes that represent the types in question.

The example in Figure 2.9 shows a first-order matching algorithm for the construction of a derivation tree for an equality judgment. Given two attributes  $tp_1$  and  $tp_2$  which values represent types (in some object language), we match *pointwise* against the structure of these types. The value of such an attribute is either the integer type constructor or a function type constructor. During evaluation, the derivation tree is constructed up to the points that the types match. The result of evaluation is an attribute *errs* that contains an error message for each type mismatch. The production *Check* does not declare any terminal nor nonterminal symbols. The example relies on higher-order children and clauses instead.

**Iteration.** A judgment  $R\bar{p}$  can be seen as a constraint  $R$  between parameters  $\bar{p}$  where  $R$  is a relation. Fixpoint iteration is often employed to gradually construct a solution to a set of such constraints.

In Figure 2.10 we show how to encode fixpoint iteration in AGs by iterating visits. We use some extensions of previous sections and Section 1.3.12 to keep the description concise. We

```

grammar Top prod Top nonterm root : Constrs -- root symbol
grammar Constrs = [Constr] -- short hand for cons-list
grammar Constr prod Subset term a, b :: Ident -- subset constraint
itf Constrs Constr -- shorthand notation
  visit solve fixed -- a fixed visit (explained below)
    chn env :: Map Ident IntSet -- chained attribute
    syn changed :: Bool -- True if env changed
    default env = head -- default rule for env
    default changed = or -- default rule for changed
sem Constr prod Subset -- approximation of loc.newVal
  loc.bVal = lookupWithDefault  $\emptyset$  loc.b lhs.env
  loc.aVal = lookupWithDefault  $\emptyset$  loc.a lhs.env
  loc.newVal = loc.aVal  $\cup$  loc.bVal
  lhs.env = insert loc.b loc.bVal lhs.env
  lhs.changed = loc.newVal  $\neq$  loc.bVal

```

Figure 2.10: AG for solving subset constraints.

explain some aspects of the example below.

We first give a grammar for a constraint language: a sentence in this language is a list of subset constraints ( $a \subseteq b$ ) on some symbols ( $a, b$ ) that represent integer sets. Given the list of constraints and an initial mapping  $env$  from symbol to integer set, we describe an algorithm that refines the mapping until all constraints are satisfied. The nonterminal  $Constr$  represents a subset constraint and the nonterminal  $Constrs$  a list of such constraints.

Secondly, we show how to refine the mapping for a single constraint, then show further below how to iterate over such a list of constraints. The rules in Figure 2.10 describe how the new  $env$  is computed from the initial  $env$  in a single iteration. The attribute  $changed$  is  $True$  if and only if the mapping was changed. The semantics for  $Constrs$  is fully determined by default rules.

Finally, we work below towards a specification of iteration for a list of constraints. The semantics of nonterminal  $Constrs$  is fully determined by the default rules: we thus specify iteration as part of the semantics of the  $Top$ -production, for which we introduce some additional annotation.

An invoke-rule may be annotated with strategies  $z$  as we saw earlier. We introduce two new strategies:  $oneshot$  and  $iterate$ . By default an invoke-rule is (implicitly) annotated as  $oneshot$ , which means that the visit is at most invoked once. However, when the annotation is  $iterate$ , then the visit may be repeated multiple times:

```

r ::= invoke x of  $\bar{c}$  z -- annotated invoke-rule
z ::= oneshot -- by default (implicit)
      | iterate e -- repetitive invocation

```

## 2 Outline of the RulerCore Concepts

*again* = *Just*     :: *Inp N x* → *Maybe (Inp N x)*   -- API function  
*stop* = *Nothing* :: *Maybe (Inp N x)*                -- API function

The expression *e* is a function that takes two parameters. The first parameter is the set of values for inherited attributes of the last iteration, and the second parameter the set of values for the synthesized attributes that resulting from that last iteration. The result of the function describes if the visit is repeated. If the result is produced using the constant *stop*, then the visit is not repeated. If, however, the result is produced using the function *again*, which takes as parameter the set of inherited attribute values that are used for the next iteration, then the visit is repeated with those values.

We apply this strategy to repeat the solve-visit on lists of constraints until a fixpoint is reached for the environment, which is the case when the attribute *changed* is *False* at the end of an iteration:

```
sem Top prod Top                                    -- semantics of the root
  inh.root.env = lhs.initialEnv                -- env for the first iteration
invoke solve of root iterate  $\lambda inp\ outp \rightarrow$  -- iterative invoke strategy
  if changed outp                                -- query attribute changed
  then again (inp {env = env outp})        -- repeat with updated env
  else stop                                     -- stop iterations
  lhs.result = syn.root.env                 -- takes result of last iteration
```

The values of the attributes are stored in a record for the inherited and synthesized attributes of a visit<sup>11</sup>. The labels are an encoding of the name of the attribute.

As a constraint solving strategy we may be interested in results of previous iterations. To keep a local state per node we introduce visit-local chained attributes, so that the notation for visit-declarations becomes:

$t ::= \mathbf{visit\ } x \overline{\mathbf{chn\ } y} :: ty \bar{r} t$    -- the type *ty* is optional

Note that the name of a visit may not clash with the name of a child, and *y* must be unique with respect to all visit-local attributes of a production.

The name *y* in the attribute declaration denotes four attributes that are local to the production:

attribute	meaning	scheduling	notation
<i>inh.x.y</i>	initial value of <i>inh.vis.y</i>	outside visit <i>x</i>	<i>inh</i> is a keyword
<i>syn.x.y</i>	last value of <i>syn.vis.y</i>	outside visit <i>x</i>	<i>syn</i> is a keyword
<i>inh.vis.y</i>	input to visit	inside visit <i>x</i>	<i>inh</i> and <i>vis</i> are keywords
<i>syn.vis.y</i>	result of visit	inside visit <i>x</i>	<i>syn</i> and <i>vis</i> are keywords

<sup>11</sup> In case of the generalization to phases of the previous section, an association of attributes to phases may be determined automatically when such an association is not manually given. However, when this happens, it is unclear which attributes are present in the record. Therefore, to be able to iterate a phase, we require the phase declaration to be annotated with the annotation *fixed* which disallows the automatic scheduling of attributes to the phase.

```

itf Constr                                     -- more visits
  visit initial   inh topVal  :: IntSet       -- value of top element
  visit solve fixed chn ...                -- as defined above
  visit generate  syn outcome :: (IntSet, IntSet) -- outcome of solving

sem Constr prod Subset
  lhs.output    = (loc.aVal, loc.newVal)       -- the computation for the result
  inh.solve.reps = 0                             -- initial val of vis chained attr
  syn.vis.last  = inh.vis.last + 1             -- increment
  ...
  loc.newVal    = loc.aVal ∪ loc.bVal ∪ loc.cVal
  loc.cVal      = if inh.vis.last ≥ 5 then lhs.topVal else ∅
  visit solve chn reps :: Int                -- visit local chained attribute

```

Figure 2.11: Example of weakening.

In Figure 2.11, we express that if the number of iterations exceeds a threshold of 5, then the result is weakened by enlarging it to the top-value in our set-lattice. This approach enforces convergence. The top-value is provided as attribute *lhs.topVal*. Note that we specified the rules outside the *solve* visit-block. The rule scheduling moves these rules to the appropriate block.

In contrast to conventional fixpoint evaluators for AGs, we precisely specify the iteration points, may perform fixpoint iteration over multiple attributes, and keep (purely functional) state between iterations. We may even construct children as part of a fixed visit, although to prevent constructing children over-and-over again, we may need to store the state of a child as part of the iteration state. We can accomplish this by detaching and attaching children.

**First-class children.** Constraints are used in inference algorithms to delegate a proof obligation to a different location in the tree. Constraints are typically used to delay a proof until all constraints in a given scope are collected.

In an AG, proof obligations can be encoded as a visit on a child that still needs to be invoked. Invoking the visit corresponds to constructing the proof. In the model as presented so far, we statically know the configuration of a child's state at each point during evaluation of the node. We present rules to detach and attach children that are in a given configuration, which permit us to treat children as first class values, and thus as constraints:

```

p = detach x of c  -- detaches c in the state that visit x is pending
attach x of c = e  -- attaches e as c in the state that visit x is pending

```

The attach-rule is a generalization of the child-rule that specifies that visit *x* and later are accessible on child *c*. The detach-rule specifies that visit *x* and later are not accessible on child *c* and provides a value that represents the child prior to the invocation of *x*. These two

## 2 Outline of the RulerCore Concepts

```
itf Expr -- some example
visit gather
  syn gathCnstrs :: IntMap Sem_Cnstrs_solve -- use to gather of children
visit distribute
  inh distCnstrs :: IntMap Sem_Cnstrs_generate -- used to distribute children
  syn transformed :: Expr
sem Expr prod Var
child c : Constr = sem.Subset loc.nm lhs.parentNm
c.topVal = lhs.topVal -- for first visit to c
lhs.gathCnstrs = singleton loc.nodeId (detach solve of c)
attach generate of c = lookup loc.nodeId lhs.distCnstrs
lhs.transformed = Expr_Const $fst $ c.output -- based on the last visit of c
```

**Figure 2.12:** Example of child detachment.

rules may be used in conjunction, however, to prevent conflicts only one attach or detach rule is allowed per child and visit combination. A detached child may be attached at a different location in the tree and visited as part of the evaluation for that location, or be visited through a wrapper function in the host language as part of an external solving algorithm.

Figure 2.12 shows an example of how a child can be detached. We collect the detached children in an attribute *gathCnstrs* as constraints. These constraints are solved elsewhere by invoking the *solve* visit on them, then transferred back as attribute *distCnstrs* and attached again. The type *Sem\_Cnstrs\_solve* is the type of a detached child prior to the invocation of *solve*. At another location in the tree, we may attach the constraints in the list and apply the iteration technique of above to solve the constraints.

This approach has the advantage that we can easily transport context information from the node that defines the constraint to the location where we solve the constraint, and vice versa. Middelkoop [2011b] gives additional examples of this technique. Moreover, the dependency analysis provides define-before-use guarantees. Chapter 7 describes how dependent types can be used to prove that a detached subtree is attached at precisely one other location in the tree.

**Remarks.** By making visits explicit we gained the ability to describe evaluation strategies by annotating the callee (visit declarations) or the caller (invoke-rules). Clauses offer a means to specify alternatives. Children in a given state are first class and can be passed around to describe complex traversals. The extensions preserve the attractive properties of AGs, such as automatic rule scheduling and purely functional descriptions. Also, the implementation in the host language is purely functional.

In Chapter 5 we present a large example, and give a translation of the notation to Haskell code. We show in Middelkoop [2011b] that we can also describe complex traversals over trees and even graph structures.



```

data Index a where      -- data type written using GADT notation
  TInt :: Index Int      -- parameterless constructor with  $a \equiv Int$ 
  TBool :: Index Bool   -- parameterless constructor with  $a \equiv Bool$ 

append :: Index a  $\rightarrow a \rightarrow a \rightarrow a$ 
append TInt  = (+)      -- coercion of Int to a
append TBool = ( $\wedge$ )  -- coercion of Bool to a

ex1 = (append TInt 1 2, append TBool True False) -- OK: (3, False)
ex2 = append TInt 1 2 + append TBool True False -- type error ( $Int \neq Bool$ )
ex3 = append TBool 1 2      -- type error ( $Int \neq Bool$ )

```

**Figure 2.13:** Example of a GADT as type index.

## 2.4 Case Study with GADTs

Constructors of an algebraic data type specify how a value of the data type is structured. A data type may be parameterized. Generalized Algebraic Data Types (GADTs) [Cheney and Hinze, 2003] associate per data constructor a set of type equivalences between the parameters of the data type. When building a value using a GADT constructor, and thus specifying how the parameters are instantiated, the type equivalences must be satisfied. In the scope of a successful pattern match against a GADT data constructor, the type equivalences may be assumed to hold and can be used to *refine* or safely *coerce* types.

In the extended edition of this thesis, we present a type system for GADTs as a case study for several reasons [Middelkoop, 2011a]. Firstly, a type system for GADTs poses additional challenges to a description of a type inference algorithm compared to a conventional DHM-style inference algorithm (Section 1.2.6, Section 1.3.11) which give insight in what features our meta language for type system needs to support. Secondly, we investigate the description of GADTs as a minimalistic type system *extension*. Moreover, we make extensive use of GADTs in this thesis, thus this chapter can then also be used as an explanation of GADTs.

In this section, we take a simplified subset of the actual type system: equality proofs. We first give a specification, then look at properties of an inference algorithm, and consider a description of the algorithm with attribute grammars.

**Specification.** GADTs are typically used as type index. In the example of Figure 2.13, the type *Index a* is a first-class description of the type *a*. By pattern matching on the description we reconstruct what the concrete type was to which *a* was instantiated. The expression *TBool* must be of type *Index Bool*, thus the *a* of *append TBool* must be *Bool*. Consequently, *ex*<sub>2</sub> and *ex*<sub>3</sub> are ill-typed.

In the above example, there exists a type equality assumption  $a \sim Int$  in the context of having matched against the *TInt* constructor. The assumption is used to prove that  $Int \rightarrow Int$  can be coerced into  $a \rightarrow a$ . The actual facilities that we need to reason with GADTs is the introduction of type equality assumptions in a scope, and equality reasoning on types. These

## 2 Outline of the RulerCore Concepts

$\boxed{\Gamma \vdash \tau \equiv \rho}$			
$\Gamma \vdash \tau \equiv \tau$ REFL	$\frac{\Gamma \vdash \tau \equiv \rho}{\Gamma \vdash \rho \equiv \tau}$ SYM	$\frac{\Gamma \vdash \tau \equiv \rho \quad \Gamma \vdash \rho \equiv \sigma}{\Gamma \vdash \tau \equiv \sigma}$ TRANS	$\frac{(\tau \sim \rho) \in \Gamma}{\Gamma \vdash \tau \equiv \rho}$ ASSUM
$\frac{\Gamma \vdash \tau \equiv \rho \quad \Gamma \vdash \sigma \equiv \omega}{\Gamma \vdash \tau \rightarrow \rho \equiv \sigma \rightarrow \omega}$ CONGR	$\frac{\tau \rightarrow \rho \equiv \sigma \rightarrow \omega}{\Gamma \vdash \tau \equiv \sigma}$ SUB.L	$\frac{\tau \rightarrow \rho \equiv \sigma \rightarrow \omega}{\Gamma \vdash \rho \equiv \omega}$ SUB.R	

**Figure 2.14:**

facilities are orthogonal to the actual treatment of algebraic data types. To be able to describe GADTs as a separate aspect of a type system, it is thus desirable to separate these facilities.

In the above example, we used simple types constructed by type arrows and type constants. In the specification we use the following grammar for types and environments containing equality assumptions:

$$\begin{aligned} \tau &::= a \mid Int \mid Bool \mid \tau_1 \rightarrow \tau_2 && \text{-- types, also: } \rho, \omega, \text{ and } \sigma \\ \Gamma &::= \emptyset \mid \Gamma, (\tau_1 \sim \tau_2) && \text{-- environment containing type equality assumptions} \end{aligned}$$

The type equality relation is used to reason with the equality between types.

Given an environment  $\Gamma$  that consists of the type equality assumptions, the inference rules in Figure 2.14 describe the type equality relation. The first three rules are properties that any equality relation is supposed to exhibit. In addition, the rule ASSUM expresses a proof by assumption, and the remaining three are related to congruence and subsumption properties derived from the structure of types. See Middelkoop [2011a] for some exemplary proofs for judgments of this relation.

**Forward and backward chaining.** The above rules are not straightforwardly mapped to an inference algorithm. The rule SYM can always be applied, thus some condition is needed to determine when not to apply this rule. In Section 1.2.6 we describes properties of the type rules of the DHM type system that permit an attractive implementation in the form of algorithm W. Similarly, we apply apply domain knowledge here to impose conditions on above rules so that problematic derivation trees are avoided or do not have to be considered by the algorithm. For example, we require that a derivation tree for judgment  $a \equiv b$  may not contain (indirectly) a child for the same judgment, as the proof would then be circular. This constraint ensures that the number of applications of the sym-rule is bounded.

To explain why the other rules are not straightforwardly mapped to an inference algorithm, we mention first that there are two ways of reasoning with inference rules [Russell et al., 1996]. With *forward chaining* inference starts with assumptions and derives conclusions. With *backward chaining* inference starts at conclusions and tries to prove premisses until

they can be discharged by assumptions. Inference algorithms as discussed so far use a limited form of backward chaining.

Backward chaining is suitable when a conclusion can be decomposed into smaller premisses, which is indeed the case for rules `REFL`, `ASSUM` and `CONGR`, and also for rule `SYM` with the aforementioned restriction. This is not the case for the rules `TRANS`, `SUB.L`, and `SUB.R`. These contain one or more meta variables in their premisses that are not fixed by their conclusion judgments. As a consequence, arbitrary (infinite) branches can be introduced in the derivation tree by applying these rules.

Forward chaining is suitable when premisses can be decomposed into smaller conclusions. This is the case for all rules except `REFL` and `CONGR`. To deal with all rules, we use a combination of backward and forward chaining by distinguishing proof obligations and proven facts. Rule `CONGR` may only be applied on a proof obligation whereas Rules `SUB.L` and `SUB.R` may only be applied on proven facts. The `TRANS` may only be applied if one of the premisses is a proven fact. As part of the case study in Middelkoop [2011a], we implemented a solver for equality constraints in UHC using an implementation of constraint handling rules [Frühwirth, 1998] that provides forward chaining and can emulate backward chaining [Dijkstra et al., 2007].

**Lookahead.** Forward chaining can be implemented with backward chaining by defining a reduction relation on environments which keep track of derived facts. We thus concern ourselves in the remainder of this section with the implementation of backward chaining using attribute grammars.

To implement backward chaining, we use clauses to represent the various alternatives. However, the rule `TRANS` poses an additional challenge: a choice made for the left premise has consequences for the right premise. To express that a clause may only be selected if the remaining evaluation in a given context (the remainder-context) does not fail, we introduce two more strategy annotations: the lookahead-strategy and the onlylocal-strategy which serve as annotations for visit-blocks and invoke-rules:

```

t ::= visit x z  $\bar{k}$       -- visit-block as presented before
r ::= invoke x of  $\bar{c}$  z  -- invoke-rule as presented before
z ::= onlylocal      -- does not take the remainder-context into account
    | lookahead     -- takes the remainder-context into account

```

The onlylocal-strategy is the default.

The remainder-context is a runtime property that can be influenced by invoke-rules. When an invoke rule is evaluated and it is annotated with the lookahead-annotation, the remaining evaluation in the parent's remainder context contributes to the remainder-context of the child. Otherwise, the invoke-rule behaves as a cut-operator which fixes the choices of clauses made by the child. The partial-strategy and total-strategy as mentioned before are orthogonal to the onlylocal-strategy and lookahead-strategy.

The example in Figure 2.15 serves as illustration and we explain it below. For brevity, we left out details of the description that are related to the prevention of infinite derivations, the administration of substitutions, and the reuse of prior derivation trees. Moreover, we

```

grammar Eq prod Check
itf Eq visit check partial
  inh env      :: Set (Ty, Ty)
  inh tp1, tp2  :: Ty
sem Eq prod Check visit check
  lookahead  -- visit annotation
  default env -- rule in scope visit
  invoke check lookahead parent
clause refl
  child u : Unify = sem_Unif
  u.tp1 = lhs.tp1
  u.tp2 = lhs.tp2
clause sym
  child flipped : Eq = sem_Check
  flipped.tp1 = lhs.tp2
  flipped.tp2 = lhs.tp1
clause trans
  child fr      : Fresh = sem_Fresh
  child left    : Eq    = sem_Check
  child right   : Eq    = sem_Check
  left.tp1    = lhs.tp1
  left.tp2    = fr.tp
  right.tp1   = fr.tp
  right.tp2   = lhs.tp2
clause assum
  match (u1.tp2, u2.tp2) ← do
    ahead $ λk → some $
      map k $ elems lhs.env
  child u1, u2 : Unify = sem_Unif
  u1.tp1 = lhs.tp1
  u2.tp1 = lhs.tp2

```

**Figure 2.15:** Example of an AG that represents an equality solver.

omitted attributes and rules to construct coercion terms from such a derivation tree. Such topics are discussed in Section 2.3. Only one production is declared for the Eq-nonterminal. The clauses in combination with higher-order children determine the structure of the equality proof.

The example features a monadic match rule. The right-hand side of this rule is a monadic expression that determines the value to match against. In clause *assum*, we take a type equality assumption from the environment. There may be multiple of such assumptions in the environment. We derive from these assumptions a monadic expression that explores the possibilities one after the other and selects the first one that succeeds. Via *ahead* (explained below), we get a continuation *k* that expects a value for the pair and performs the remaining computations for the current context. The function *some* is defined below. It selects the first computation that succeeds.

We saw above how to express backward chaining with clauses in combination with lookahead. In Section 2.1 we mentioned that the AG can be expressed as a monad. First we show an implementation of lookahead by using a backtracking monad, then show how clauses can be mapped to this monad.

**Backtracking monad.** We wrap the actual underlying monad *m* into a monad transformer *BackT* that consists of a composition of the continuation transformer on top of the error transformer. The continuation monad transformer provides a continuation, and via the error monad transformer a failing computation can be observed [Jones, 1995]. The result type of the continuation is the parameter *r*:

```

data Back = Back -- backtrack message
type BackT r m a = ContT r (ErrorT Back m) a -- transformer

```

Backtrack points are specified using the operator  $(\oplus)$  which represents local choice. It selects its right argument if and only if the evaluation of the left argument fails. Alternatively, the operator  $\oplus$  represents a global choice, which takes the continuation of the parent of the choice into account:

```

( $\oplus$ ) :: Monad m => BackT a m a -> BackT a m a -> BackT r m a
p  $\oplus$  q = ContT (\lambda c -> catchError (cut p) (const (cut q))) >>= c)
( $\oplus$ ) :: Monad m => BackT r m a -> BackT r m a -> BackT r m a
p  $\oplus$  q = ContT (\lambda c -> catchError (runContT p c) (const (runContT q c)))
cut p    = runContT p return
msum    = foldr ( $\oplus$ ) (fail "backtrack")
resolve p = ContT (\lambda c -> cut p >>= c)

```

The function *resolve* limits the continuation. The function *ahead* exposes the continuation to the higher-order function *f*:

```

ahead :: Monad m => ((a -> ContT r m r) -> ContT r m r) -> ContT r m a
ahead f = ContT (\lambda c -> runContT (f (\lambda a -> ContT (\lambda k -> c a >>= k))) return)
p  $\oplus$  q = ahead (\lambda k -> p >>= k  $\oplus$  q >>= k) -- alternative implementation

```

The function *ahead* provides the ability to explore different values for the continuation, and make choices based on the outcome of the continuation. We show in Chapter 6 how to extend this mechanism to make choices based on intermediate results that are computed in the continuation.

In a continuation monad, a computation *BackT r m a* represents a computation for a value of type *r* with a pending computation that takes *a* to *r*. The function *f* in *ahead* takes the pending computation as parameter, and replaces the computation for *r* with a computation that immediately goes to *r*. *Ahead f* can thus be understood as replacing the pending computation with (the computation produced by) *f*.

**Mapping of clauses to monads.** The evaluation algorithm for a clause is a monadic expression that computes values for the synthesized attributes of the visit. We thus define the body of a visit function as a sequence of these monadic expressions that are either combined with the global choice operator when the visit is annotated with the lookahead-annotation, or with the local choice operator when the visit has the default onlylocal-annotation. If an invoke-rule is not annotated with a lookahead-annotation, it applies *resolve* to the monadic expression of the child after applying the values for the inherited attributes.

**Remarks.** As mentioned in the previous section, clauses represent a search tree, which encodes alternative ways to compute the decorations of the tree. The exploration of these alternatives using the *BackT* monad is depth-first. Chapter 6 describes how to explore clauses in a breadth-first way, which may give a more balanced exploration.

## 2 Outline of the RulerCore Concepts

Overhead is the work that is performed for the exploration of an alternative that is not selected. In practice, we preferably solve problems using a single pass traversal, or a fixpoint iteration. A search for a solution, however, cannot always be avoided, as is demonstrated by the GADT use-case. Moreover, the naive exploration of alternatives may be convenient for prototyping purposes.

### 2.5 Attribute Grammars with Stepwise Evaluation

Some type inference algorithms require an exploration of a forest of potential derivation trees. We can encode such a forest as a search tree that contains additional nodes which represent choices between derivations. In Chapter 6 we present a library to describe such explorations of the search tree.

**Stepwise evaluation.** In the evaluation algorithms of Chapter 5, clauses are explored one after the other. This approach corresponds to a depth-first exploration of alternatives. In Chapter 6 we show how to evaluate clauses simultaneously, which corresponds to a breadth-first exploration of alternatives. A breadth-first exploration provides a balanced exploration for alternatives, which may be more efficient.

With statically ordered AG evaluation (Section 1.3.4), the evaluation of an AG is a sequence of rule evaluations. In this section, we group a number of these rule evaluations together and call that a step. We represent the evaluation of a tree as a computation which can be asked to execute one step, and afterwards pauses and returns control back to the caller. To decorate the tree, we provide a computation (the root-computation) at the root which takes the computation of the tree and repeatedly asks it to perform a step until the decorations are computed.

**Simultaneous exploration.** To explore alternatives, we mentioned in the previous section that we combine the computations of alternatives, for example using the  $\oplus$ -operator. We now consider different ways to combine the computations of alternatives. We provide a computation (the choice-computation) that asks the alternatives to perform steps in an interleaved fashion. When an alternative succeeds, we replace the choice-computation with the alternative. When an alternative fails, we replace the choice-computation with the other alternative. When each alternative performed one step, the choice-computation exposes one step to its parent choice-computation or the root. With this approach we obtain a breadth-first traversal of alternatives.

In this section, we first describe how to write such an algorithm as a monad that represents a *coroutine*, and how to specify what constitutes to a step in this monad. Then, we show how this monad is used in RulerCore descriptions, and show an implementation of the monad.

**Coroutines.** A coroutine is a function that during its execution performs zero or more *yield* operations which denote re-entry points. A *yield* operation pauses the execution of the function and returns control to the caller. The caller may resume the execution of the callee from the point where it was paused. The callee may expose intermediate results to the caller

```

yield :: Stepwise m ()
step  :: Monad m => Stepwise m a -> m (Report m a)
lift  :: Monad m => m a -> Stepwise m a
ahead :: (forall r. (a -> Stepwise m r) -> Stepwise m r) -> Stepwise m a

data Report m a          -- represents a progress report
= Done a                 -- finished and produced a value a
  | Failed String        -- failed with a given error message
  | Paused (Stepwise m a) -- paused with the residual computation

```

**Figure 2.16:** API of the Stepwise monad.

and the caller may provide additional parameters when resuming the function. We assume initially as simplification that no results are exchanged between caller and callee.

Visit functions are examples of coroutines (Section 1.3.5) that are invoked a statically fixed number of times. The evaluation of a child pauses at the end of the visit, and proceeds with the evaluation of the next visit when the parent invokes the subsequent invoke-rule. In this section, however, we consider coroutines that in addition to the statically fixed yields between visits, may yield a statically unbounded number of times during the execution of a visit. The evaluation up-to the next yield is what we call a step.

We design a coroutine monad *Stepwise*, which represents a stepwise computation specialized for the exploration of alternatives. It supports a number of operations in addition to those of the *BackT* monad. Figure 2.16 shows the API. The operation *yield* pauses the execution and resumes the caller. The operation *step* runs the coroutine until the coroutine either fails or succeeds, or reaches the next yield instruction. The outcome of the evaluation is presented as a progress report in the encapsulated monad *m*. A yield-operation thus specifies what constitutes as a single step. With *lift*, we wrap the effects of *m* into a stepwise computation so that these effects can be merged with the effects embedded in other stepwise computations. For example, we typically use *lift* to describe how the effects of a step-operation on a child are merged with some parent computation.

With the choice combinators, we define a computation that represents a traversal over a search tree. Each subtree encodes an alternative. With the above API, Figure 2.17 shows a breadth-first version of the choice combinators. The traversal is breadth-first because when *act* reports a step to the caller, each of the non-failed children performed one step. Iteration is encoded by replacing the choice-computation with a computation that calls the choice function again. Thus, when we commit to a certain alternative, we replace the choice-computation with the selected alternative, and thereby eliminate the choice.

Figure 2.18 A depth-first version of the choice combinators is obtained by applying the function *fullred* to the left alternative. This function returns the computation with all steps stripped, thus forcing it to evaluate fully. The control we have over stepwise computations allows us to express a whole range of strategies, such as taking two steps left for each step right.

## 2 Outline of the RulerCore Concepts

$p \oplus q = ahead (\lambda k \rightarrow p \ggg k \oplus q \ggg k)$	
$p \oplus q = \mathbf{do} a \leftarrow lift (step\ p)$	-- perform a step for $p$
$b \leftarrow lift (step\ q)$	-- perform a step for $q$
$act\ a\ b$	-- inspects the outcomes
$act :: Report\ m\ a \rightarrow Report\ m\ a \rightarrow Stepwise\ m\ a$	
$act\ (Done\ a)\ \_ = return\ a$	-- commit to finished $p$
$act\ \_ (Done\ a) = return\ a$	-- commit to finished $q$
$act\ (Failed\ s)\ (Failed\ \_) = fail\ s$	-- both fail
$act\ (Failed\ \_)\ (Paused\ r) = r$	-- $p$ fails, commit to $q$
$act\ (Paused\ r)\ (Failed\ \_) = r$	-- $q$ fails, commit to $p$
$act\ (Paused\ p')\ (Paused\ q') = yield\ \ggg\ (p' \oplus q')$	-- pause, later continue with choice

**Figure 2.17:** Breadth-first choice combinators.

$fullred :: Stepwise\ m\ a \rightarrow m\ (Stepwise\ m\ a)$	
$fullred\ p = \mathbf{do}\ rep \leftarrow step\ p$	-- perform a step
$\mathbf{case}\ rep\ \mathbf{of}$	-- inspect report
$Paused\ r \rightarrow fullred\ r$	-- repeat after yield
$\_ \rightarrow return\ \$\ comp\ rep$	-- either <i>Failed</i> or <i>Done</i>
$comp :: Report\ m\ a \rightarrow Stepwise\ m\ a$	-- report to residual computation
$comp\ (Paused\ m) = m$	
$comp\ (Failed\ s) = fail\ s$	
$comp\ (Done\ v) = return\ v$	

**Figure 2.18:** Depth-first choice combinators

The underlying monad  $m$  can be used to exchange information between the computation of an alternative and the choice between alternatives. For example, when  $m$  is a writer monad, an alternative can provide an estimate of the amount of work that has been performed. When  $m$  supports IO, the system time can be used to balance the two computations. Stepwise computations thus offer a means to describe powerful and complex exploration strategies.

**Children as stepwise computations.** The evaluation of an AG we represent as a monadic computation, and thus fits the Stepwise-monad straightforwardly. In the remainder of this section, we show how to specify yield operations and how to express alternatives.

We do not need to introduce additional syntax to express yield operations because monadic child rules have monadic right-hand sides and can therefore be used to express the yield operations. For example, we introduce a dummy nonterminal *Yield* in Figure 2.19 and use it to specify a yield operation using a child rule in some exemplary production *Var*. A monadic



```

itf Yield                                -- interface without visits nor attributes
grammar Yield prod Yield                 -- single production
sem Yield prod Yield                     -- empty semantics
sem Expr prod Var visit check
  child y:Yield ← do                       -- monadic child rule
    yield                                    -- monadic operation
    return sem_Yield                         -- semantics of child (trivial)

```

Figure 2.19: Yielding of steps expressed as nonterminal.

```

sem Tree prod Alt                          -- semantics for a choice node
  child left :Tree = ...                    -- define child left
  child right :Tree = ...                  -- define child right
  left.i1 = ...                            -- definition of some attributes
  right.i1 = ...
  loc.p = detach upon v of left             -- evaluates left up to visit v
  loc.q = detach upon v of right           -- evaluates right up to visit v
  attach upon v of res : N ← do            -- attaches as child res
    loc.p ⊕ loc.q                           -- choice between children
  lhs.s = res.s1                           -- use results of the chosen child

```

Figure 2.20: A sketch of a parallel exploration.

child rule is guaranteed to be evaluated in the visit it is constrained to. Thus, the right-hand side of the rule is evaluated during visit *check*.

**Encoding of alternatives.** To express alternatives we can explicitly encode a search tree using higher-order children or by using clauses. We first consider the encoding of a search tree. A node in a search tree may express a choice between its children. For this purpose we refine the notation introduced in Section 2.3 to attach and detach children with an additional keyword (explained below).

We detach alternatives and attach a computation that determines the chosen alternative. The abstract example in Figure 2.20 provides a sketch. Recall that the semantics of a visit of a child is a function that takes values for inherited attributes and returns a monadic computation for the synthesized attributes of that visit. We are thus interested in the state of the children after they received the values for the inherited attributes. We use the upon-keyword for this purpose. The detach-rule thus provides the monadic computation for the specified visit of a child, and the attach-rule runs the computation to obtain the synthesized values for that visit.

```

data Stepwise m a where
  Return  :: a → Stepwise m a
  Bind    :: Stepwise m a → Parents m a b → Stepwise m b
  Fail    :: String → Stepwise m a
  Yield   :: Stepwise m ()
  Lift    :: m a → Stepwise m a
  Ahead   :: (∀r.(a → Stepwise m r) → Stepwise m r) → Stepwise m a

data Parents :: m a b where
  Root    :: Parents m a a
  Pending :: (a → Stepwise m b) → Parents m b c → Parents m a c

instance Monad (Stepwise m) where
  return  = Return
  m >>= f = Bind m (Pending f Root)

```

**Figure 2.21:** The structure of *Stepwise*.

**Strategy for clauses.** Instead of explicitly encoding a search tree as above, we can also use clauses. To specify a choice between clauses, we present additional notation. A visit-block may be annotated with a select-strategy:

```

t ::= visit x z  $\bar{c}$     -- existing syntax for visit-blocks
z ::= onlylocal      -- combine clauses with the local choice operator
    | lookahead     -- combine clauses with the global choice operator
    | select e       -- custom, e is e.g. a function  $\lambda c_1 \dots c_k \rightarrow \dots$ 

```

The select-strategy specifies a function  $e$  that takes a computation for each clause of the visit-block as parameter and provides a computation for the results of the visit. The computation for the results of a visit is  $e\ c_1 \dots c_k$  where  $c_1, \dots, c_k$  are the computations corresponding to each clause<sup>12</sup>.

**Implementation.** In the remainder of the section we describe how intermediate results can be exposed to the selection function. However, we first show how the Stepwise-monad is implemented. The algorithm<sup>13</sup> that we show here is slightly simplified with respect to Chapter 6 and its full understanding is not required for the remainder of this chapter.

We represent the Stepwise-monad in Figure 2.21 as a computation that can be inspected (Section 1.3.9). The function *step* interprets the computation in order to evaluate it one step. The right-hand side of a bind contains a stack *Parents* of all the continuations to the right of a monadic expression. The expression  $(m \gg=f) \gg=g$  is represented as:

<sup>12</sup> Several notational variants are possible. It may sometimes be more convenient to obtain the computations of the clauses as a list or as a record with a field for each clause.

<sup>13</sup> Complete Haskell module of the simplified implementation:  
<https://svn.science.uu.nl/repos/project.ruler.papers/archive/RefStepwise.hs>

$step :: Monad\ m \Rightarrow Stepwise\ m\ a \rightarrow m\ (Report\ m\ a)$	
$step\ m$	$= reduce\ m\ Root$
$reduce :: Monad\ m \Rightarrow Stepwise\ m\ a \rightarrow Parents\ m\ a\ b \rightarrow m\ (Report\ m\ b)$	
$reduce\ Yield\ r$	$= return\ \$\ Paused\ (r\ 'apply'\ ())$
$reduce\ (Fail\ s)\ -$	$= return\ \$\ Failed\ s$
$reduce\ (Lift\ m)\ r$	$= m\ \gg\ \gg\ step.\ apply\ r$
$reduce\ (Ahead\ f)\ r$	$= step\ \$\ f\ (apply\ r)$
$reduce\ (Return\ v)\ Root$	$= return\ \$\ Done\ v$
$reduce\ (Return\ v)\ (Pending\ f\ r)$	$= reduce\ (f\ v)\ r$
$reduce\ (Bind\ m\ r)\ r'$	$= reduce\ m\ (push\ r\ r')$
$apply :: Parents\ m\ a\ b \rightarrow a \rightarrow Stepwise\ m\ b$	
$apply\ r\ v$	$= Bind\ (Return\ v)\ r$
$push :: Parents\ m\ a\ b \rightarrow Parents\ m\ b\ c \rightarrow Parents\ m\ a\ c$	
$push\ r\ Root$	$= r$
$push\ Root\ r$	$= r$
$push\ (Pending\ f\ r')\ r$	$= Pending\ f\ (push\ r'\ r)$

**Figure 2.22:** The implementation of *reduce*.

$Bind\ m\ (Pending\ f\ (Pending\ g\ Root))$

Since monadic binds are right-associative, the stack only grows when an expression occurs as left-and side of a monadic bind that expands to one or more binds. This is the case when calling the visit function of a child, hence the stack contains the continuations of all parents till the location where *step* is performed. The child that is undergoing evaluation is on top of the stack.

The function *reduce* in Figure 2.22 takes a computation *m* and a pending stack *p*. It evaluates *m* one step. If *m* yields or fails it returns a progress report. Otherwise, it continues evaluating *m* until it obtains a result that can be fed into the parents-stack. If the parents-stack is empty, evaluation is finished. Otherwise, the parent-stack contains the continuation to proceed with. The function *step* delegates to *reduce* with an empty stack. The function *apply* turns a pending parent into a monadic computation by passing it the result it was waiting for. The function *push* concatenates two parent stacks.

**Coordination.** We presented above how to specify selection strategies. To encode powerful coordination strategies, we improve the above approach so that computations can yield results and take arguments when resumed.

**Definition (Tag).** A tag of type *Op i o* specifies the interface between the callee (the computation) and the caller (execution of *step*), where *i* is a type index that fixes the data-type that is used for tags (*Op i o* is a type family), and *o* is the type of the results exchanged between

## 2 Outline of the RulerCore Concepts

```

action :: Op i o → Inp o → Stepwise m (Out o)
data Report i m a      -- refinement of the Report data type
  | ∀o. Paused (Op i o) (Inp o)
              ((Stepwise m (Out o) → Stepwise m a) → Stepwise m a)
data family Op i :: * → * -- a tag of an operation (of some set indexed by i)
type family Inp o :: * -- inputs to the operation Op i o
type family Out o :: * -- resulting outputs of operation Op i o

```

**Figure 2.23:**

callee and caller. The computation yields results of the type *Inp o* and takes arguments of type *Out o* as parameter for the next visit.

We refine the operation *yield* so that it takes a tag of type *Op i o* and intermediate results of type *Inp o* for the caller and provides arguments of type *Out o* as given by the caller when the callee is resumed. Figure 2.23 shows the encoding in Haskell with type families.

For example, *action* can be used by a computation to report the number of open goals, and receive a priority rating from the caller. In that case, we specify the following instances for the above type families:

```

data Meta                -- type index for the set of tags
data OInfo               -- type index for a particular tag
data instance Op Meta o where -- declares the tags
  OpInfo :: Op Meta OInfo      -- one tag
type instance Inp OInfo = Int  -- specification of inputs of operation o
type instance Out OInfo = Int  -- specification of results of operation o

```

The report-handling code in selection strategies may match on Yield-reports to obtain the *Op i o* and *Inp o* values, plus the continuation which may be used to resume the computation when a computation is provided that produces the *Out o* values.

This approach is powerful: arbitrary *traps* to operations can be expressed this way. For example, it is possible to create tags for that represent operations such as unification, generation of unique numbers, lookups in memo tables, and output to the console. The caller can determine what semantics to give to these operations. The construction offers an *inversion of control*: the callee *declaratively* specifies operations, and the caller determines the semantics.

**Memoization.** As mentioned in the GADT example, it may be desirable to cache results of subtrees and reuse these results at other locations in the tree. However, if the continuation (accessed using *ahead*) is used to distinguish the result of the computation, the results should not be cached, unless the continuation is the same for each context the shared computation appears in.

Moreover, in case of a stepwise computation, these results may not be available yet, and it may be desirable to share computations instead. A computation can be shared by storing it in an updatable state, and updating this state after each call to *step*. The computation may then each time receive the result of an action from a different context. Also, each contexts may only receive a partition of the actions yielded by the computation.

**Remarks.** Chapter 6 focusses on *generators* which are coroutines that only yield information but do not take parameters. We show how to evaluate such coroutines strictly or lazily. In the latter case, results can already be produced when it depends on a choice for which only one alternative is left.

## 2.6 Attribute Grammars with Dependent Types

In Chapter 7 we investigate AGs where attributes may have a dependent type, which can be used to state and prove properties of the AG. For this purpose, we describe an embedding of AGs in Agda<sup>14</sup> [Bove and Dybjer, 2009]. *Dependent types* provide a means to use types to encode properties with the expressiveness of (higher-order) intuitionistic propositional logic, and terms to encode proofs. In this setting, a parameterized type constructor specifies a relation between its type parameters and data constructors form the inference rules of the relation.

In a dependently typed AG, the type of an attribute may refer to values of attributes. The type of an attribute is an invariant and the value of an attribute a proof for that invariant. Moreover, because of the Curry-Howard correspondence, dependently typed AGs are a domain-specific language to write structurally inductive proofs in a *composable, aspect-oriented* fashion; each attribute represents a separate aspect of the proof.

Some knowledge of dependent types and Agda is a prerequisite for this section. We first give an example and some notation. We follow up with an extension that permits the visits of a nonterminal to be organized as a tree instead of a totally ordered sequence.

**Dependent attribute type.** The following interface declaration for some nonterminal *Pat* demonstrates attributes with a dependent type. The gathered environment *syn.gathEnv* is a subset of the final environment *inh.fnEnv*, which is expressed as the type  $\text{syn.gathEnv} \subseteq \text{inh.fnEnv}$ . We assume the existence of a type constructor  $\subseteq$  and several utility functions. The attribute *inh.gathInFin* has the above type and therefore represents a proof of this property:

```

itf Pat
  visit analyze   syn gathEnv  :: Env
  visit translate inh fnEnv    :: Env
                               inh gathInFin :: syn.gathEnv  $\subseteq$  inh.fnEnv
grammar Pat prod Var term nm :: Ident

```

<sup>14</sup> In this section, we deviate slightly from the actual syntax to have a closer correspondence with Haskell.

## 2 Outline of the RulerCore Concepts

The *gathInFin* attribute provides the guarantee that elements that are in *syn.gathEnv* are also in *inh.finEnv*. Rules of productions of *Pat* may exploit this guarantee.

The *lookup* of an identifier in the final environment may return *Left notIn* where *notIn* is a proof that the identifier is not in the environment, or *Right v* where *v* is the value of the identifier in the environment.

In the following example, production *Var*, which has a terminal *loc.nm*, we define with a proof<sup>15</sup> *loc.prv<sub>2</sub>* that the identifier is in the environment, and we use *loc.prv<sub>2</sub>* to prove that the lookup cannot return a Left-value:

### sem *Pat* prod *Var*

<i>loc.prv<sub>1</sub></i> = <i>here loc.nm syn.lhs.gathEnv</i>	-- proof of <i>nm</i> in <i>gathEnv</i>
<i>loc.prv<sub>2</sub></i> = <i>inSubset lhs.gathInFin loc.prv<sub>1</sub></i>	-- proof of <i>nm</i> in <i>finEnv</i>
<i>loc.val</i> <b>case</b> <i>loc.nm</i> 'lookup' <i>lhs.finEnv</i> <b>of</b>	-- defined by case distinction
<i>Left notIn</i> <b>falsum</b> <i>notIn loc.prv<sub>2</sub></i>	-- impossible case
<i>Right v</i> → <i>v</i>	-- case that <i>loc.nm</i> is in <i>finEnv</i>

The case that the element is not in *lhs.finEnv* is in contradiction with *loc.prv<sub>2</sub>*. Their application has an uninhabitable type, which we use in combination with the *falsum*-case to terminate the branch without giving a definition.

This example shows three ways to define an attribute: with a plain RHS, with case distinction, and with **falsum** *e* where host-language expression *e* has an uninhabitable type:

<i>r</i> ::= <i>p = e</i>	-- with plain RHS
<i>p m</i>	-- with complex RHS
<i>m</i> ::= <b>falsum</b> <i>e</i>	-- unreachable case ( <i>e</i> has an uninhabitable type)
<b>case</b> <i>e of</i> $\bar{b}$	-- with case distinction, and cases $\bar{b}$
<i>b</i> ::= $\rho \rightarrow e$	-- nested plain RHS in a case distinction
$\rho t$	-- nested complex RHS in a case distinction
$\rho$	-- pattern in the host language (no attributes)

The additional syntax provides us with a means to give provable *total* definitions of attributes.

**Cycle analysis and consistency.** Functions are required to be total in dependently typed programs for reasons of logical consistency and termination of type checking, which in case of AGs correspond to total definitions of attributes and the requirement that dependencies between attributes are acyclic.

**Partitions.** In Chapter 4, the ordering algorithm implicitly distinguishes different contexts in which a nonterminal is used. However, to ensure that attribute definitions are total, it may be convenient to make such contexts explicit.

In the following example, we specify code generation depending on the absence of errors. We define two contexts for the *generate* visit. The context *errorfree* provides an attribute

<sup>15</sup> The functions *here* and *inSubset* are conventional dependently typed functions that construct the appropriate proofs. Their implementation are beyond the scope of this section.

```

sem Pat prod App
  lhs.errors = f.errors ++ a.errors           -- collect errors
  context errorfree                          -- rules exclusive for errorfree
    invoke generate of left context errorfree -- specifies context to invoke
    invoke generate of right context errorfree -- specifies context to invoke
    left.noErrors = leftNil  left.errors right.errors lhs.noErrors
    right.noErrors = rightNil left.errors right.errors lhs.noErrors
    lhs.code = left.code `apply` right.code
  context haserrors                          -- rules exclusive for haserrors
    invoke generate of left context haserrors
    invoke generate of right context haserrors
    lhs.pretty = left.pretty ⊕ right.pretty    -- collect pretty print

```

**Figure 2.24:** Example of rules specified for a specific context.

*code*, but it may only be invoked when errors are absent. The context *haserrors* alternatively provides an attribute *pretty*, which contains an annotated pretty print of the program:

```

itf Pat
  visit report syn errors :: Errs inh.finEnv
  visit generate           -- a visit may consist of one or more partitions
  context errorfree       -- a partition has a name
    inh noErrors :: syn.errors ≡ []
    syn code           :: Target inh.finEnv
  context haserrors      -- a partition may also contain subsequent visits
  syn pretty           :: Doc

```

The caller invokes a visit on the callee, and is responsible for selecting what context it wants to use. The callee is required to produce results for that choice. The callee can encode restrictions on the available choices for the parent as inherited attributes. The caller must provide values for the inherited attributes of the partition it chooses.

We specify a context as annotation of the invoke-rule. Moreover, we may specify rules for particular contexts as is demonstrated in Figure 2.24. A special *falsum*-rule may be used to denote that a visit, clause or context is unreachable.

**Remarks.** Type attributes correspond to quantification. An inherited type attribute corresponds to universal quantification, since the caller can choose its instantiation. A synthesized type attribute corresponds to existential quantification. The callee can choose its type, but the caller cannot make an assumption about it. This mechanism allows us to deal with polymorphism in interfaces.

Indeed, the above ideas allow quantification in an AG for Haskell to be expressed. In a dependently typed AG, attributes can represent both values and types. In Haskell, there is a

## 2 Outline of the RulerCore Concepts

clear distinction between values and types. In an AG for Haskell, we can make an explicit distinction between attributes that represent types (and specify a *kind* as type) and attributes that represent values. The type of a type attribute may not refer to other attributes. The type of a value attribute, however, may refer to a type attribute.

## 2.7 Attribute Grammars on DAGs

In the extended edition, we included a relatively short chapter [Middelkoop, 2011a] that provides examples of other ways to apply the techniques as presented. There are two common data structures in compilers: trees and directed graphs. Ordered attribute grammars are suitable to define a semantics on trees but not suitable to define the semantics of graphs. The reason is that nodes in a graph may occur in different contexts at execution time, which makes a static dependency analysis difficult. In that chapter, we also show how our approach relates to (cyclic) reference attribute grammars.

## 2.8 Conclusion

We gave a detailed summary of the following chapters, and described how the chapters are connected together. Also, this chapter showed the features of RulerCore in relation to conventional attribute grammars.

A prototype implementation of RulerCore is available as the compiler `ruler-core`. Its implementation is based on higher-order attribute grammars and Haskell, and can be obtained from:

```
https://svn.science.uu.nl/repos/project.ruler.papers  
/archive/ruler-core-1.0.tar.gz
```

The `examples` subdirectory contains some minimalistic examples. A large example based on the HML type system [Leijen, 2009] can be obtained from:

```
https://svn.science.uu.nl/repos/project.UHC.pub  
/branches/tnfchris-hml/
```



## 3 AGs with Side Effects

This chapter introduces the concept of visits, which play an important role in subsequent chapters of this thesis. We present this concept by means of a correspondence with the visitor design pattern.

The visitor design pattern is often applied to describe traversal algorithms over Abstract Syntax Trees (ASTs) in imperative programming languages. It defines a *visitor*, an object with a visit method that is executed for each node in the AST, and updates the state of the visitor, and possibly the states of nodes as well. The order in which the visitor visits the nodes is explicitly under control of the programmer, which is essential to deal with the side-effectful computations that modify the state of the visitor. However, the exchange of results between traversals is error-prone.

Attribute grammars with a statically ordered attribute evaluation (Section 1.3.5) are an alternative way to describe multi-traversal algorithms. An Attribute Grammar (AG) defines attributes of nodes in the AST as functions of other attributes, and an attribute evaluator decorates the AST with the attributes in one or more traversals. The attributes form a convenient mechanism to exchange results between traversals. A strong point of AGs is that the order of evaluation is implicit. As a consequence, however, AGs discourage the use of side effects.

We present RulerCore, a language that combines attribute grammars with visitors. In RulerCore, sufficient assumptions can be made about the evaluation order to facilitate side effects. In Chapter 4 we show how to formally reason with such side effect.

A RulerCore grammar can be used in combination with several host languages. In the outline of this chapter (Section 2.1) we sketched RulerCore with the purely functional, statically typed language Haskell as host language. In this chapter, we actually show RulerCore in combination with the imperative and dynamically typed language JavaScript<sup>1</sup>. This chapter thus introduces the concepts that underly the subsequent chapters without a dependency on knowledge of Haskell. Also, it serves as a basis of how contents of the subsequent chapters can be mapped to other languages than Haskell.

### 3.1 Introduction

Algorithms for traversing tree-shaped data structures appear in many applications, especially in compilers. A lot of effort has been invested in developing proper abstractions for tree traversals, for example in the form of a tree-walking automaton (Section 1.3.3), or in a more abstract way with Attribute Grammars (AGs) [Knuth, 1968].

---

<sup>1</sup> In the outline of this chapter, we limited side effects in RulerCore to rules that determine the shape of children. Since we cannot enforce the absence of side effects in JavaScript expressions, we do not impose this restriction. Instead we present a pin-rule, which can be restricted to a visit and allows for safe use of side effect.

### 3 AGs with Side Effects

AGs are an attractive language for the development of compilers. We applied AGs in many small projects (to teach compiler construction [Utrecht, 2010], master projects, etc.), and several large projects, including the Utrecht Haskell Compiler [Dijkstra et al., 2009], the Helium [Heeren et al., 2003b] compiler, and the editor Proxima [Schrage and Jeuring, 2004]. AGs are an important asset in these projects. The example in Section 3.2 demonstrates some of the reasons.

Tree traversals play a role in many other fields, including end-user applications. Web applications, for example, traverse and compute properties of DOM trees. Unfortunately, the abstractions that emerge from research in compiler construction are not used to write such traversals. To use AGs, sufficient familiarity with the formalism is required, which may be an obstacle for many programmers. Also, tool support is typically absent for the programming language in question, and the AG formalism poses severe restrictions to be used effectively in these areas, such as prohibition of side effect. In this chapter, we treat the latter two challenges, which are of a technical nature.

Considering the first challenge, for imperative languages like JavaScript, a programmer either writes recursive functions, or takes a more structured approach via the visitor design pattern [Gamma et al., 1993, Palsberg and Jay, 1998, Oliveira et al., 2008]. Tool support for the visitor design pattern is available for many languages. For example, the parser generator SableCC [Gagnon and Hendren, 1998] generates visitor skeleton code for the ASTs that the parser produces. With visitors, side effects are used to carry over results computed in one visit to the next visit. In our experience, the scheduling of visits and their side effects is an error-prone process, due to the absence of the define-before-use guarantee. We elaborate on this in Section 3.2.1.

Attribute grammars offer a programming model where each node in the AST is associated with named values that are called *attributes*. An AG description contains computations that define attributes in terms of other attributes. If these definitions are noncircular, the description can be translated to a multi-visit traversal algorithm where each attribute is defined before it is used. The scheduling of the computations is implicit, which saves a programmer from writing the scheduling manually, and thus also cannot do it wrong. However, the implicit scheduling comes with a severe restriction: side effects cannot be used reliably and should not be used in attribute computations. In web applications, for example, we typically need side effects to influence the contents of a webpage. We elaborate on this in Section 3.2.2.

The main contribution of this Chapter is an extension of attribute grammars that has an explicit notion of visits, which offers a hybrid model between visitors and attribute grammars, while maintaining the best of both worlds. In fact, besides being more expressive, our extension make attribute grammars more intuitive to use.

We also address the second challenge, which is to make our approach available for many host languages. We present RulerCore, a small but powerful language for tree traversals. We managed to isolate the language-dependent part into a small subset called RulerBack, and show the translation from RulerBack to JavaScript. In later chapters, we show a translation to Haskell. With these two languages, we cover the implementation issues regarding the full spectrum of mainstream general purpose programming languages available today.

Similar to other preprocessed languages, code fragments of the host language are embedded in RulerCore to describe the computations for attributes. The embedding keeps general-

purpose programming constructs out of RulerBack, and allows the programmer to express computations without having to learn a special language. In particular, RulerBack is suitable as a host language for attribute grammars.

In this chapter, we present the languages RulerCore and RulerBack. We do so by using an example based on the alignment of HTML menus. This example requires a traversal of the AST to determine the sizes of the HTML items, and another pass to compute the locations of the items. Section 3.2 presents the example in each of the above languages.

This chapter focussed on RulerBack. We introduce RulerBack in Section 3.3 and show a translation to JavaScript in In Section 3.4. In Section 3.5 we get back to RulerCore and describe the translation to RulerBack.

## 3.2 Example

In this section, we motivate the claims of the introduction in more detail, and introduce the background information relevant for the remainder of the chapter. We take as a use case the alignment of an HTML menu in a web application using JavaScript, based on a multi-visit tree traversal over an abstract description of the menu. We first show a solution using the visitor-pattern, then a near-solution using attribute grammars, and finally two solutions using RulerCore.

### 3.2.1 Visitor Design Pattern

In the visitor design pattern, each node of the Abstract Syntax Tree (AST) is modelled as an object, which stores references to the subtrees, and has an *accept* method. The *accept* method takes a visitor as parameter. A visitor is an object with a *visit*-method for each type of node. The *accept* method of the AST node calls the appropriate *visit*-method on the visitor and passes the node as an argument. This *visit* method consists of statements that manipulate the state of the visitor and the AST node, and can visit a subtree by calling the *accept* method on the root of a subtree, with the visitor-object as parameter.

Figure 3.1 shows an example of a visitor that lays out HTML items as a menu in a tree-like fashion, as visualized in the upper-right corner of the figure. The menus are aligned to the right, and submenus are slightly indented. Furthermore, we desire the items to have a minimal size, but large enough to contain their contents. The variable *root* contains an abstract description of the menu as a tree of *Menu* objects (the AST). Associated with each *Menu* object is an HTML item with the same name. We interpret the menu structure to layout the HTML items. In the first visit to the menu tree, we query the widths of the corresponding HTML items. In the second visit, we adjust the positions and sizes of these items. Some information (such as indentation based on the *depth*) is computed in the first visit, and also needed in the second visit. This information is stored as additional fields in the menu objects.

The order in which the tree is visited is clearly defined by the explicit *accept*-calls in the *visit*-methods. The order of the calls ensures that the sized of the HTML items are queried before they are resized.

### 3 AGs with Side Effects

```

function Menu (name, children) {
  this.name = name;
  this.children = children;
}

Menu.prototype.accept = function (visitor) {
  visitor.visitMenu (this);
}

function Visitor () {
  this.depth = 0;
  this.maximum = 0;
  this.count = 0;
}

var root =
  new Menu ("a", [
    new Menu ("b", [
      new Menu ("c", [])
    ]),
    new Menu ("d", [])
  ]);
]; -- <div id="anchor" onLoad="align(root, this);"></div>

function align (root, anchor) {
var v = new Visitor ();

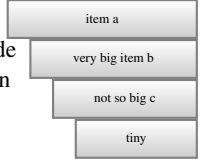
v.visitMenu = function (menu) {
  menu.elem = document.getElementById (menu.name);
  menu.depth = this.depth;
  this.maximum = Math.max (this.maximum, this.depth * 20 + menu.elem.clientWidth);
  for (var i in menu.children) {
    this.depth = menu.depth + 1;
    menu.children [i].accept (this);
  }
}

root.accept (v);

v.visitMenu = function (menu) {
var offset = menu.depth * 20;
  menu.elem.style.left = (anchor.offsetLeft + offset) + "px";
  menu.elem.style.top = (anchor.offsetTop + this.count * 30) + "px";
  menu.elem.style.width = (this.maximum - offset) + "px";
  menu.elem.style.height = 30 + "px";
  this.count++;
  for (var i in menu.children) {
    menu.children [i].accept (this);
  }
}

root.accept (v);

```



**Figure 3.1:** Pseudocode of dual-visit menu alignment.

However, there are a number of issues with the above solution. In the second visit, we require that a number of values are computed in the first visit. These values are stored in the state of the AST nodes during the first visit. This approach has a number of problems. It does not guarantee that the values that we store indeed those values that we need later. Furthermore, we never remove any of these values from the state, and thus retain all memory until the AST gets deallocated. This especially becomes a problem when using large ASTs in which many results are stored.

Furthermore, the order of appearance of the statements is relevant. For example, the value *this.depth* needs to be reset at the appropriate place, and requires that the assignment to *menu.depth* is done before. Similarly, the increment to *this.count* needs to be positioned carefully. These are actually separate aspects that we would like to implement in isolation. However, separate pieces of code cannot easily be composed due to side effect.

Finally, we need to explicitly write visits to children using *accept*. Some tools generate depth-first visitors, which alleviates the need to do so. However, such approaches come with restrictions. The restriction that all statements must take place before the invocations to children is an example. In Figure 3.1 we reset *this.depth* in between visits to children. To use a depth-first visitor, we would have to move this statements, which may not be immediately possible. Moreover, in the simple example that we showed, the two visits are invoked after each other at the root. In practice, for example in type checking languages with principal types, we actually invoke multiple visits on a subtree before moving on to the next subtree. This rules out depth-first visitors, and is also error-prone to write manually.

The example in Figure 3.1 can be made more complicated by allowing menus to share submenus. The menu structure then forms an acyclic directed graph instead of a tree. With such a complication, the problems mentioned above become harder to deal with.

As a sidenote, in this chapter, we treat the AST as a fixed data structure. For example, we do not consider adding menu entries on the fly. The ideas we propose can deal with the dynamic construction of proof trees (Chapter 5), and we think that this is sufficient to deal with dynamic changes to the AST as well, but leave this topic as future work.

Below, we look for a way to generate code similar to the code above, but using a description that alleviates the programmer from the aforementioned problems.

### 3.2.2 Attribute Grammars

Attribute grammars take care of the problems mentioned above related to visitors, but are not flexible enough to take side effects into account. Before we show the example, we first give some background information on attribute grammars, and their encoding in JavaScript.

We introduced attribute grammars in Section 1.3.1, and we use a similar syntax here with minor differences due to the JavaScript host language and to stay close to the syntax that we introduce later with respect to RulerCore. To summarize, an attribute grammar is an extension of a context-free grammar. Nonterminals are annotated with attributes. Productions specify equations between attributes. The context-free grammar specifies the structure of the AST. Each node of the AST is associated with a production, and thus also the nonterminal of the nonterminal symbol that appears as left-hand side of the production. Each child of a node corresponds to a nonterminal symbol on the right-hand side of the production.

### 3 AGs with Side Effects

For example, we can denote a production as well as the structure of a node in the AST using a grammar definition (explained below):

```
grammar Menus                                -- nonterminal Menus
  prod Cons hd:Menu tl:Menus                 -- production Cons, with two nonterminals
  prod Nil                                     -- production Nil, empty
```

This grammar definition introduces a nonterminal *Menus* with two productions, representing a cons-list. The first production is named *Cons*. In BNF notation, it corresponds to  $Menus \rightarrow Menu\ Menus$ . The two nonterminals *Menu* and *Menus* in the right-hand side (RHS) have explicitly been given the respective names *hd* and *tl*. Terminals only have a name (shown later in Figure 3.2).

The grammar declaration corresponds to generated JavaScript constructor functions in the host language, which can be used to construct ASTs. Each production is mapped to a constructor function that gets as parameter an object corresponding to the symbols in the RHS of the production. Each nonterminal is mapped to a constructor function that creates a base object that each of the objects corresponding to the productions inherits. Because of inheritance, we can verify at the point of construction that the AST matches the grammar:

```
function Menu () {}                                -- nonterminal Menu: base class
function Menus_Cons (hd,tl) {}                 -- production Cons: subclass
  this.hd = hd; assert (hd instanceof Menu);
  this.tl = tl; assert (tl instanceof Menus);
}
Menus_Cons.prototype                             = new Menu ();
Menus_Cons.prototype.constructor                 = Menus_Cons;
function Menus_Nil () {}                         -- production Nil: subclass
Menus_Nil.prototype                             = new Menu ();
Menus_Nil.prototype.constructor                 = Menus_Nil;
```

Cons-lists occur often in AGs. As a shortcut, the following shorthand notation may be used, which specifies that the nonterminal *Menus* is a list of *Menu* nonterminals:

```
grammar Menus: [Menu]
```

This shorthand notation has an additional benefit: the list of menus is conceptually a cons-list in the AG description but represented efficiently as a JavaScript array in the generated code. This distinction is hidden from the programmer.

The evaluation of an attribute grammar constitutes to running an evaluation algorithm on each node. The algorithm is derived from the equations of the production that is associated with the node. The algorithm describes the decoration of the node with attributes. We assume that attributes are physically represented as JavaScript properties of the AST objects. Nodes are decorated with two types of attributes: inherited attributes are computed during evaluation of the parent of that node, and synthesized attributes are computed during evaluation of the node itself.

We declare the attributes of a nonterminal using an attribute declaration:

```
attr Menu inh depth    -- inherited attribute
      syn gathMax      -- synthesized attribute
```

These attribute names are mapped to object properties named *\_inh\_depth* and *\_syn\_gathMax*. At some point during attribute evaluation, given a participating *Menu* object *m*, the objects properties *m.\_inh\_depth* and *m.\_syn\_gathMax* will be defined. An inherited attribute may have the same name as a synthesized attribute: they are mapped to differently named properties. As an aside, nodes may define a number of local attributes, which can be seen as local variables.

To give a semantics to these attributes, we organize equations (rules) per production in *semantics-blocks*. We explain the following example below:

```
datasem Menu  -- nonterminal Menu
prod Menu    -- production Menu
  cs:depth    = 1 + lhs:depth          -- rule
  loc:width   = 20 * lhs:width         -- rule
  lhs:gathMax = Math.max (loc:width,cs:gathMax) -- rule
```

The full details of the nonterminal and its semantics can be found in Figure 3.2.

The left-hand side of an equation designates an attribute. The notation for attribute occurrences *nodename:attrname* refers to an attribute *attrname* of some node *nodename*, where *nodename* is either the name of a child, or *loc* or *lhs*. The colon ensures that attribute occurrences are distinct from JavaScript notation for properties. Attribute occurrences in the left-hand side of a rule refer to inherited attributes of children, but a synthesized attribute of *lhs* and a local attribute in case of *loc*. Thus, the attributes we need to define appear as left-hand side. For example, the above attribute occurrences refer to the JavaScript properties *this.cs.\_inh\_depth*, *this.\_loc\_width*, and *this.\_syn\_gathMax* respectively.

Similarly, the right-hand side consists of a JavaScript expressions, with embedded attribute occurrences. In this case, we may refer to the synthesized attributes of children, or with *lhs* to the inherited attributes of the current node. The terminals of a production are available as local attributes. In production *Menu*, there is a terminal called *name*, which is available as attribute *loc:name*. The translation of attribute references is similar as described above. For example, the last rule expands to the JavaScript statement:

```
this._syn_gathMax = Math.max (this._loc_width,this.cs._syn_gathMax);
```

Evaluation of an attribute grammar corresponds to traversing the AST one or more times, and executing rules, according to an evaluation strategy. In this chapter, we restrict ourselves to the class of well-defined attribute grammars, whose attribute dependencies can be statically proven to be acyclic [Knuth, 1968]. For these grammars, the attributes can be computed by visiting each node a bounded number of times. This corresponds precisely with typical uses of the visitor-design pattern.

```

grammar Root prod Root root:Menu           -- node with a child named root
grammar Menu prod Menu name cs:Menus        -- node with a property name, and a child cs
grammar Menus: [Menu]                       -- conceptually a cons-list, physically an array
var root = new Root_Root (                  -- the Menus are physically represented
  new Menu_Menu ("a", [                     -- as an array. However, conceptually
    new Menu_Menu ("b", [                 -- we define its attributes using the
      new Menu_Menu ("c", [])             -- above cons-list representation.
    ], new Menu_Menu ("d", []))));
attr Menu Menus inh depth finMax count     -- gathMax: width of submenu
  syn gathMax count                         -- note: count is both inh and syn
function align (root, anchor) {            -- uses embedded attribute grammars
  datasem Root prod Root                   -- equations of production Root of nont Root
    root:depth = 0                          -- initial depth
    root:count = 0                          -- initial count
    root:finMax = root:gathMax              -- choose gathered max as global max
  datasem Menu prod Menu                   -- production Menu of nonterm Menu
    cs:depth = 1 + lhs:depth                -- increase depth for submenus
    cs:count = 1 + lhs:count               -- increase count
    lhs:count = cs:count                   -- provide the updated count to the parent
    loc:elem = document.getElementById (loc:name)
    loc:offset = lhs:depth * 20             -- indentation
    loc:width = loc:offset + loc:elem.clientWidth
    lhs:gathMax = Math.max (cs:gathMax, loc:width)
    cs:finMax = lhs:finMax                  -- pass down final maximum
    loc:dummy = ( function () {            -- side-effectful statements (wrapped)
      loc:elem.style.left = (anchor.offsetLeft + loc:offset) + "px";
      loc:elem.style.top = (anchor.offsetTop + lhs:count * 30) + "px";
      loc:elem.style.width = (lhs:finMax - loc:offset) + "px";
      loc:elem.style.height = 30 + "px"; } ) () -- unwrap directly
  datasem Menus prod Cons                   -- equations of production Cons
    hd:depth = lhs:depth                    -- pass depth downwards through the menus
    tl:depth = lhs:depth
    hd:count = lhs:count                    -- thread the count through the menus, in an
    tl:count = hd:count                     -- in-order fashion. First to the head, then to
    lhs:count = tl:count                    -- the tail, then back up to the parent.
    lhs:gathMax = Math.max (hd:gathMax, tl:gathMax)
    hd:finMax = lhs:finMax                  -- pass global maximum downwards
    tl:finMax = lhs:finMax
  datasem Menus prod Nil                   -- equations of production Nil
    lhs:count = lhs:count                  -- thread count through without changing it
    lhs:gathMax = 0                         -- initial maximum
  var inhs = new Inh_Root ();                -- contains inh attrs of the root (empty)
  eval_Root (sem_Root, root, inhs);        -- run the attribute evaluator

```

**Figure 3.2:** Attribute grammar-based near-solution to menu alignment.



From a semantics-blocks (datasem-blocks in Figure 3.2), a function is generated that contains the evaluation algorithm. For example, the function *sem\_Menu* is generated from the semantics of nonterminal *Menu*. Furthermore, to interface with the decorated tree in JavaScript code, a function *eval\_Menu* is generated that takes the AST, the function *sem\_Menu*, and an object containing values for the inherited attributes. It applies the semantic value and returns an object with the synthesized attributes:

```

var inhs = new Inh_Menu ();
inhs.depth = 0;                                -- provide inh attrs of root
syns = eval_Menu (sem_Menu, menu, inhs);       -- initiate evaluation
window.alert (syns.gathMax);                  -- access syn attrs of root

```

In Figure 3.2, we show an attribute grammar version of the example that we presented earlier. It is a non-solution, for reasons explained later, but exhibits various important properties. Below, we comment on some aspects of the example.

The attribute grammar code in Figure 3.2 starts with a number of grammar definitions that describe the structure of the menu tree. We then define a number of attributes. In particular, the idea is that we gather a maximum *gathMax* (synthesized), and use its value at the root to pass down the global maximum *finMax* (inherited). Moreover, we count the menus. The inherited attribute *count* specifies the count for the current menu, and the synthesized *count* is the count incremented with the total number of children.

We define the semantics for these attributes in the function *align*. Because *root* and *anchor* are its parameters, we also have access to these in the right-hand sides of rules.

A HTML item can be laid out using statements that assign to properties of an HTML item. Since the right-hand side of an attribute equation (rule) is an expression, a sequence of statements needs to be wrapped as an expression. In JavaScript, this can be accomplished in a variety of ways. In the example, we choose to use a parameterless anonymous function for this purpose.

In the semantics of *Menus*, rules are given to compute the attributes for lists of menus using the cons-list representation. These rules follow standard patterns. The attributes *depth* and *finMax* are passed topdown. The attribute *gathMax* is computed bottom-up. The attribute *count* is threaded through the tree. In the visitor-example, the fields in the visitor combined with side effects took care of this behavior. With attribute grammars, we have to describe it explicitly. However, with copy rules (Section 1.3.12), collection rules [Magnusson et al., 2007], and a generalization called default rules (Chapter 5), we can abstract from these patterns, so that a more concise semantics of *Menus* can be given (as we see later).

The AG code has three nice properties. Firstly, the order of appearance of the rules is irrelevant. This allows the rules for *depth* and *count* to be written separately and merged automatically [Löh et al., 1998]. In the example, we give all the rules without using such composition facilities. However, for larger projects the ability to write such rules separately is important with respect to modularity.

Secondly, a nice property is the absence of invocations of visits (the *accept* calls in the visitor-example). The number of visits is totally implicit. From the dependencies between attributes in the rules, it can be determined automatically that the attribute *root*:*gathMax*

### 3 AGs with Side Effects

(in the semantics of *Root*) must be computed in a visit before the visit where it is passed as *root:finMax*.

Thirdly, we check statically if there is an evaluation order of statements such that all attributes are defined before their value is accessed. The attribute declarations describe the attributes that must be defined, and those that are available. The rules describe what attributes must be available before computing an attribute, and an evaluation order is possible if the transitive closure of the dependencies is acyclic [Knuth, 1968].

Unfortunately, when the above is evaluated on-demand it is incorrect because the order of evaluation of rules is determined is not only determined by dependencies on attributes but also by the side effects that rearrange the HTML items. Since the latter effects are not present as a dependency between rules and attributes, the order of evaluation may be wrong. In fact, the root of the tree does not have any attributes defined, so when assuming a on-demand evaluation of the grammar, it is actually expected that none of the rules are evaluated. Hence, we allow the programmer to explicitly encode the dependencies imposed by side effects in the next section.

#### 3.2.3 RulerCore

We now present a solution using RulerCore in Figure 3.3 which resembles the code in Figure 3.2. We discuss similarities and differences below.

The essential difference is that RulerCore has notation to explicitly describe visits to an AST node during attribute evaluation, and notation to associate side effects with individual visits.

**Interfaces.** Instead of declaring attributes for a nonterminal, we declare an *interface* for a nonterminal. An interface declaration specifies the visits of a nonterminal and attributes per visit. The following example specifies that the attributes of *Menu* are computed in two visits:

```
itf Menu                -- interface for nonterminal Menu
visit gather           -- declaration of first visit
syn gathMax            -- synthesized attr computed by visit
visit layout           -- declaration second visit
inh finMax count       -- two inherited attributes
syn count              -- synthesized attr computed by visit
```

The order of appearance of visit declarations dictates the order of visits to AST nodes with this interface. In order to visit a node, all previous visits must have occurred. Values for inherited attributes must be provided prior to the visit. Values for synthesized attributes are only available after a visit has been performed.

**Scheduling.** The rules of a semantics-block are automatically scheduled over visits using an as-late-as-possible strategy (Section 3.5.2). If the rules are cyclic, the scheduling is not possible, and a static error is reported. The scheduling determines which children to visit and in what order. However, since *Root* has no attributes, there is no need to invoke any visits of

```

grammar Root prod Root root:Menu -- node with a child named root
grammar Menu prod Menu name cs:Menus -- node with a property name, and a child cs
grammar Menus: [Menu] -- conceptually a cons-list, physically an array

var root = new Root_Root ( -- the Menus are physically represented
  new Menu_Menu ("a", [ -- as an array. However, conceptually
    new Menu_Menu ("b", [ -- we define its attributes using the
      new Menu_Menu ("c", []) -- above cons-list representation.
    ,new Menu_Menu ("d", [ [] ])]));

if Root visit perform -- root node has one visit, but no attrs
if Menu Menus -- if for nonterminals Menus (menu nodes)
  visit gather inh depth syn gathMax -- first visit: compute maximum
  visit layout inh finMax count syn count -- second visit: layout the HTML items

function align (root, anchor) { -- uses embedded attribute grammars
  datasem Root prod Root -- equations of production Root of Root
    root:depth = 0 -- initial depth
    root:count = 0 -- initial count
    root:finMax = root:gathMax -- global max is the gathered max here
    invoke layout of root -- require that visit layout of root is invoked
  datasem Menu prod Menu -- equations scheduled to visits of Menu
    cs:depth = 1 + lhs:depth -- increase depth for submenus
    cs:count = 1 + lhs:count -- increase count
    lhs:count = cs:count -- provide the updated count to the parent

    loc:offset = lhs:depth * 20 -- indentation
    loc:width = loc:offset + loc:elem.clientWidth
    lhs:gathMax = Math.max (cs:gathMax, loc:width)
    cs:finMax = lhs:finMax -- pass down final maximum

  visit gather
    pin loc:elem = document.getElementById (loc:name)
  visit layout -- equations for visit layout and later
    pin _ = (function () { -- side-effectful statements (wrapped as function)
      loc:elem.style.left = (anchor.offsetLeft + loc:offset) + "px";
      loc:elem.style.top = (anchor.offsetTop + lhs:count * 30) + "px";
      loc:elem.style.width = (lhs:finMax - loc:offset) + "px";
      loc:elem.style.height = 30 + "px";
    }) () -- directly call the anonymous function
  datasem Menus -- standard patterns for Menus
    default depth = function (depths) { return depths [depths.length - 1]; }
    default finMax = function (maxs) { return maxs [maxs.length - 1]; }
    default gathMax = function (maxs) { return Math.max.apply (Math, maxs); }
    default count = function (counts) { return counts [0]; }
    prod Cons -- each production must be explicitly listed,
    prod Nil -- even if they do not have individual rules

  var inhs = new Inh_Root_perform (); -- contains inh attrs for the root (empty)
  eval_Root (sem_Root, root, inhs); -- run the attribute evaluator

```

Figure 3.3: RulerCore solution to menu alignment.

### 3 AGs with Side Effects

*root*. Therefore, we specify through an invoke-rule that visit *layout* must be invoked, which requires through attribute dependencies that also visit *gather* must be invoked, and kickstarts the evaluation.

**Scheduling constraints.** Rules can be constrained to visits. Rules that appear in a visit-block are constraint to that visit or a later visit. The example below illustrates the various possibilities. An attribute definition that is prefixed with the keyword **pin** is restricted to exactly the visit that it appears in, and is executed during that visit even where there are no value dependencies on the attributes that it defines:

```
dataseq Menu                -- rules for nonterminal Menu
prod Menu                  -- rules for production Menu
  cs:count = lhs:count + 1  -- scheduled in visit gather or later
visit gather
  pin loc:elem = ...        -- precisely in visit gather
  visit layout              -- rules for visit layout or later
    pin _ = ...              -- precisely in visit layout
    lhs:count = cs:count   -- constrained to layout or later
```

With an underscore, we bind the value of the RHS of a rule to an anonymous attribute that we cannot refer to anymore.

A visit-block may contain rules and optionally either a nested visit-block or a nested clause-block. We use and explain clause-blocks later.

A visit-block introduces a subscope. A local attribute defined in a visit-block is not available for a rule defined in a higher scope, even if that rule is scheduled to a subscope. Attributes of children are available to higher scopes.

After all these preparations, we finally present the RulerCore solution in Figure 3.3. In this example, we express that the side effects that query the widths of the HTML items are constrained to the first visit, and that the side effects that change the locations and dimensions are constrained to the second visit.

For the *Menus*-nonterminal, we give default-rules for equality named attributes in its productions. If such an attribute (e.g.  $k_i:a$ ) does not have an explicit definition, it is implicitly defined by the default rule. Associated with the children in a production is their order of appearance. The default-rule provides a function which receives a list (an array in JavaScript) as argument that contains the values of the attributes  $a$  of the children in the production and preserving the order of the children. Children without an attribute  $a$  do not have a value of this list. Also, the value of inherited  $lhs:a$  is added to the end of the list if it exists.

**Remarks.** In the above example, we combined both side effects and attribute evaluation. We retain the advantages that AGs offer, such as the ease of adding attributes. As we show later, the description still permits the AG to be analyzed and the rules to be ordered.

However, we require the programmer to manually assign attributes to visits, and to constrain side-effectful rules to particular visits, which is not necessary for conventional attribute grammars. In practice, this is only a minimal amount of extra work that has as an additional advantage that it makes attribute evaluation more predictable and thus easier to understand.

```

function align (root, anchor) {
  var sem_Root =
    sem prodRoot : Root
    visit perform
      clause Root
        child root : Menu = sem_Menu
        root : ast = lhs : ast
        root : depth = 0
        root : count = 0
        root : finMax = root : gathMax
        invoke layout of root
  var sem_Menu =
    sem prodMenu : Menu
    visit gather
      clause Menu
        child cs : Menus = sem_Menus
        cs : ast = lhs : ast.cs
        cs : depth = 1 + lhs : depth
        pin loc : elem = document.getElementById (loc.name)
        loc : offset = lhs : depth * 20
        loc : width = loc : offset + loc : elem.clientWidth
        lhs : gathMax = Math.max (cs : gathMax, loc : width)
        cs : finMax = lhs : finMax
        visit layout
          clause Menu'
            cs : count = 1 + lhs : count
            lhs : count = cs : count
            pin _ = (function () {
              loc : elem.style.left = (anchor.offsetLeft + loc : offset) + "px";
              loc : elem.style.top = (anchor.offsetTop + lhs : count * 30) + "px";
              loc : elem.style.width = (lhs : finMax - loc : offset) + "px";
              loc : elem.style.height = 30 + "px";
            }) ()
  ... -- See Figure 3.5
  var inhs = new Inh_Root_perform ();
  inhs.ast = root
  eval_Root_perform (sem_Root, inhs);
}

```

**Figure 3.4:** Desugared RulerCore solution to menu alignment (part 1).

### 3.2.4 Desugared RulerCore

In Figure 3.4 (explained below), we give a different *desugared* of Figure 3.3. Both versions are valid RulerCore programs. This desugared version only uses a subset of RulerCore, which we call *RulerBack*. This representation is more verbose, but more suitable for code generation.

RulerBack generalizes over higher-order [Vogt et al., 1989] and conditional [Boyland, 1996] attribute grammars. In the next section, we introduce RulerBack. The example in Figure 3.4 serves as preparation. In Figure 3.4, we omit the grammar definitions, interface declaration, and *root* variable, which are equal to those in the first half of Figure 3.3.

In an attribute grammar, there is a fixed association between a node in the AST and a production and a fixed association between a production and a collection of rules. The code to execute for a node in the AST is derived from the associated collection of rules. RulerBack *virtualizes* productions: we define grammars that describe traversals instead of data structures (Section 1.3.12). The rules of a RulerBack production are organized in *clauses* (introduced below), and rules can programmatically determine which clauses to evaluate.

The above functionality allows us to define a single production per nonterminal. The non-terminal has an inherited attribute *ast* which contains the AST as an inspectable value. Note that in RulerBack the representation of cons-lists using arrays becomes visible whereas this is hidden in the RulerCore example. In the translation from RulerCore to RulerBack additional RulerBack rules are generated to treat the explicit array representation. We show this in the example.

*Semantics blocks*, which are of the form **sem**  $P:N\dots$ , introduce a production  $P$  of nonterminal  $N$ . The visits and attributes of  $N$  are declared separately with an *interface declaration*. Additionally, the code generated from a sem-block is a constructor-function that produces an AST node with attributes as described by  $N$ . The AST is provided explicitly as the inherited attribute *ast*.

In Figure 3.4, we start with a definition of the semantics for the root. The interface *Root* declares one visit, and we give rules for that visit in a visit-block. RulerBack provides clauses as a means to generalize over productions. Each clause provides a way to compute the attribute values of a visit. Moreover, a clause may specify constraints. Clauses are executed in the order of appearance. A clause is selected if its constraints are satisfied. Conventional productions, which specify a constraint on the node of the tree, can thus be represented with clauses.

Clauses and visit-blocks may contain rules. Rules given for a visit are in scope of all clauses declared for that visit. Rules for a clause are only visible in that clause. We introduce child-rules. A child-rule introduces child. In the example, we introduce a child *root*, with interface *Menu*, and the semantics defined by the JavaScript value *sem.Menu*. This is an example of a higher-order child (Section 1.3.7) and is used to virtualize the AST. Unlike in later chapters, in this chapter the virtual AST is isomorphic to the actual AST. Also, we assume that a visit  $v$  to child  $x$  is only possible if there exists an invoke-rule for it.

The left-hand sides of an evaluation-rule may be a pattern. This is either an attribute reference, an underscore, or a constant. Evaluation of such a rule fails when its execution throws an exception or the left-hand side is a value that is not equal to the value computed for

```

function align (root, anchor) {
    ...
    var sem_Menus =
        sem prodMenus : Menu
        visit gather
            default depth = function (depths) { return depths [depths.length - 1]; }
            default finMax = function (maxs) { return maxs [maxs.length - 1]; }
            default gathMax = function (maxs) { return Math.max.apply (Math, maxs); }
            default count = function (counts) { return counts [0]; }
        clause Cons
            match true = lhs : ast.length ≥ 1
            child hd : Menu = sem_Menu
                hd.ast = lhs : ast [0]
            child tl : Menu = sem_Menus
                tl.ast = lhs : ast.slice (1)
        clause Nil
    var inhs = new Inh.Root_perform ();
    inhs.ast = root
    eval_Root_perform (sem_Root, inhs);

```

**Figure 3.5:** Desugared RulerCore solution to menu alignment (part 2).

the right-hand side. Such a failing rule causes an exceptional termination of the evaluation, unless the evaluation-rule is prefixed with the match-keyword, and the rule does not throw an exception other than a special fail-exception. In this case, we say that the clause *fails*. Thus, the match-rules allow us to distinguish clauses *Cons* and *Nil* of *ntMenus* by matching on the length of the list.

Invoke rules and visit-blocks may be annotated with strategies of various kinds. Chapter 2 describes these strategies: in this chapter the strategies **partial** and **total**, which describe backtracking behavior, appear.

During attribute evaluation, the clauses of a visit are evaluated in the order of appearance. When evaluation for a clause fails, the evaluation *backtracks* to the next clause<sup>2</sup>. A backtrack does not revert potential side effects of the results that are evaluated so far. If the last clause fails, the default behavior is that the evaluation fails exceptionally. However, if both the visit itself and the invoke-rule of the parent are annotated with the strategy **partial**, then the invoke-rule of the parent fails and causes backtracking in the parent. By default, a total strategy is assumed.

Unspecified visit-blocks are implicitly defined as an empty visit-block. A visit-block without clauses implicitly has a single clause. This clause matches always unless match-rules are present. Therefore, we neither have to specify the visit-block *layout* nor clauses for it in the

<sup>2</sup> Chapter 6 describes a different strategy where clauses are simultaneously tried.

$e ::= \mathcal{J} [\bar{b}]$	-- embedded RulerBack blocks $b$ in JavaScript code $\mathcal{J}$
$b ::= i \mid s \mid o$	-- RulerBack blocks
$i ::= \mathbf{itf} \ I \ v$	-- interface decl, with visit sequence $v$
$v ::= \mathbf{visit} \ x \ \mathbf{inh} \ \bar{x}_1 \ \mathbf{syn} \ \bar{x}_2 \ z \ v$	-- visit decl, with attributes $x_1$ and $x_2$ and strategy $z$
$\square$	-- terminator visit (optional in the notation)
$s ::= \mathbf{sem} \ x : I \ t$	-- semantics expr, defines production $x$
$t ::= \mathbf{visit} \ x \ \bar{r} \ \bar{k}$	-- visit-block, with rules $r$ and clauses $\bar{k}$
$\square$	-- no visit (serves as terminator)
$k ::= \mathbf{clause} \ x \ \bar{r} \ t$	-- clause definition, with next visit $t$
$r ::= p = e$	-- assert-rule, evaluates $e$ , binds to pattern $p$
$\mathbf{pin} \ p = e$	-- pinned assert rule (bound to visit it occurs in)
$\mathbf{match} \ p = e$	-- match-rule, backtracking variant
$\mathbf{invoke} \ x \ \mathbf{of} \ \bar{c} \ z$	-- invoke-rule, invokes visit $x$ on $\bar{c}$
$\mathbf{child} \ c : I = e$	-- child-rule, introduces a child $c$ of itf $I$
$z ::= \mathbf{partial} \mid \mathbf{total}$	-- behavior in case of rule failure
$o ::= c : x$	-- attribute reference in some embedded code
$p ::= c : x$	-- attribute reference in pattern
$-$	-- wildcard
$K$	-- constant $K$
$x, c, p, e$	-- identifiers, child identifiers, patterns, expressions respectively
$\Gamma, \Sigma ::= \varepsilon$	-- attr+child environment (used in semantics)
$\Gamma, \circ$	-- new scope
$\Gamma, \mathbf{inh} \ c : x$	-- inh attr $c : x$
$\Gamma, \mathbf{syn} \ c : x$	-- syn attr $c : x$
$\Gamma, c : I \ v$	-- child $c$ with available visit sequence $v$
$\Phi ::= \varepsilon$	-- interface environment (used in semantics)
$\Phi, I \ v$	-- itf $I$ with visit decls $v$

Figure 3.6: Syntax of RulerBack



```

itf S visit  $v_1$  inh l syn  $\emptyset$  total      -- decompose array l down
  visit  $v_2$  inh  $\emptyset$  syn s total      -- compute sum s up
  □                                             -- end of visit decls

var sumArr = sem sum : S

  visit  $v_1$   $\emptyset$                                 -- first visit
    clause sumNil                                -- when list is empty
      match  $0 = lhs:l.length$                     -- match empty l
        visit  $v_2$   $\emptyset$                         -- second visit
          clause sumNil2                        -- single clause
            lhs : s = 0                            -- empty list, zero sum
            □                                       -- end of visit blocks

        clause sumCons                            -- when list non-empty
          loc : x = lhs : l [0]                  -- head of the list
          loc : xs = lhs : l.slice (1)          -- tail of the list
          child tl : S = sumArr                -- recursive call
          tl : l = loc : xs                    -- l param of call
          invoke  $v_1$  of tl total              -- invoke on child

          visit  $v_2$   $\emptyset$                         -- second visit
            clause sumCons2                    -- single clause
              invoke  $v_2$  of tl total        -- invoke on child
              lhs : s = loc : x + tl : s    -- sum of head and tail
              □                                       -- end of visit blocks

```

Figure 3.7: Example of RulerBack syntax: summing an array of integers.

semantics of *ntMenus*. Also, because of the automatic ordering of rules, many of the rules defined in visit *layout* of *ntMenu*, could also be defined one level higher, in visit *gather*.

Note that this representation is more general than conventional attribute grammars, and that an attribute grammar can easily be mapped to this representation, as shown by the difference between Figure 3.3 and Figure 3.4.

### 3.3 Static Semantics of RulerBack

In this section, we introduce RulerBack, a small subset of RulerCore. It serves as an intermediate language for RulerCore. Figure 3.6 shows the syntax of RulerBack. A RulerBack program *e* is a JavaScript program  $\mathcal{J}$ , with embedded RulerBack blocks *b*. A block *b* is either an interface declaration, semantics-block, or attribute reference. The syntax of visits in interface declarations and semantics-blocks use a cons-list representation which is convenient for the specification of translation schemes later. We explain the individual forms of syntax in more detail below.

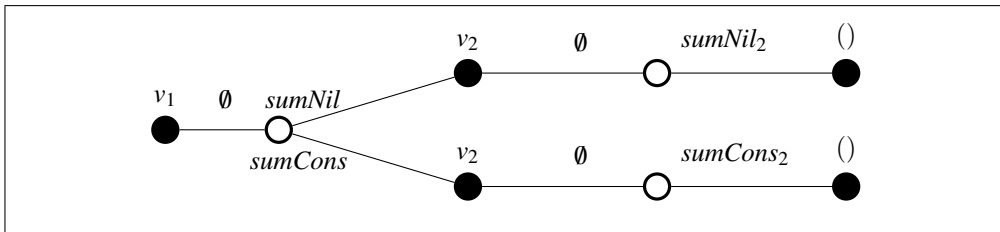
### 3 AGs with Side Effects

There are some essential differences with respect to RulerCore that we gradually introduced in the previous section. The order of appearance of rules defines the evaluation order, and each invocation of a visit must explicitly be stated through an invoke rule. Grammar blocks can be desugared and are optional in RulerBack. Instead, with clauses and (match) rules, we provide a general mechanism to traverse arbitrary JavaScript data structures.

The embedded blocks may occur anywhere in a JavaScript program. The programmer is required to position semantics blocks and attribute references at expression positions in the host language, and interface declarations at statement positions. It is the responsibility of the programmer to handle the scoping of embedded blocks.

Figure 3.7 shows a RulerBack program that computes the sum of an array of integers in two visits. This simple example can also be formulated as a single visit. However, it serves here as a short example of a dual-visit program. The first visit has two clauses: a clause *sumNil* when the array is empty, and *sumCons* when there is at least one element. In the second visit, the actual sum is computed, using the rules that depend on which clause is chosen in the first visit.

A semantics-block introduces a visitor-object with an interface *I*. The interface dictates what visits can be made to the object, and what the inputs (inherited attributes) and outputs are (synthesized attributes). The outputs for a visit are produced by executing rules. We write these rules down in a tree of clauses and visits, as illustrated by the indentation in Figure 3.7 and the state diagram in Figure 3.8.



**Figure 3.8:** States of nodes with semantics *sum*.

The black nodes represent the state of the AST-node prior to a visit and the white nodes indicate a branch point. Upon creation, an AST node is in the state represented by the root node. With each edge, alternatively the rules of a visit or the rules of a clause are associated. With each visit, an AST node changes state to a next black node by executing the rules on the path to such a node. Execution of all of the rules must succeed. At a branch-point, rules on edges of clauses are tried in order of appearance. Results produced by executing rules are in scope of rules further along the path.

There are four types of rules in RulerCore.

- **match**  $p = e$             -- match-rule
- match**  $loc : x = 3$         -- example that succeeds
- match**  $true = false$       -- example that fails

The pattern  $p$  must match the value of the right hand side. If the evaluation of  $e$  results in an exception, or the match fails, a backtrack is made to the next clause. If  $p$

$\Gamma \vdash s$	$v; \Gamma \vdash t$	$\Sigma; \Gamma_0 \vdash r: \Gamma_1$	$\Gamma \vdash e$	-- signatures of the relations
$\Gamma \vdash o$	$v; \bar{x}; \Gamma \vdash c$	$\Sigma; \Gamma_0 \vdash p: \Gamma_1$		-- used by judgments below
$\frac{x \text{ unique} \quad Iv \in \Phi \quad v; \Gamma, \circ \vdash t}{\Gamma \vdash \mathbf{sem} x: I t} \text{ SEM} \quad \square; \Gamma \vdash \square \text{ END}$				
$\frac{\Gamma_0 \cup \mathit{avail}(\mathbf{visit} x \bar{r} \bar{k}); \Gamma_0 \cup \{\mathbf{inh} lhs: a \mid a \in \bar{i}\} \vdash \bar{r}: \Gamma_1 \quad \overline{v; \bar{s}; \Gamma_1 \vdash \bar{k}}}{\mathbf{visit} x \mathbf{inh} \bar{i} \mathbf{syn} \bar{s} v; \Gamma_0 \vdash \mathbf{visit} x \bar{r} \bar{k}} \text{ VISIT}$				
$\frac{x \text{ unique} \quad \Gamma_0 \cup \mathit{avail}(\mathbf{clause} x \bar{r} \bar{k}); \Gamma_0 \vdash \bar{r}: \Gamma_1 \quad v; \Gamma_1 \vdash t \quad \{\mathbf{syn} lhs: a \mid a \in \bar{s}\} \subseteq \Gamma_1}{v; \bar{s}; \Gamma_0 \vdash \mathbf{clause} x \bar{r} t} \text{ CLAUSE}$				
$\frac{\Sigma; \Gamma_0 \vdash p: \Gamma_1 \quad \Gamma_0 \vdash e}{\Sigma; \Gamma_0 \vdash \mathbf{pin}^? p = e: \Gamma_1} \text{ ASSERT} \quad \frac{\Sigma; \Gamma_0 \vdash p: \Gamma_1 \quad \Gamma_0 \vdash e}{\Sigma; \Gamma_0 \vdash \mathbf{match} p = e: \Gamma_1} \text{ MATCH}$				
$\frac{\Phi(I_c) = \bar{v} \quad \mathbf{visit} x \mathbf{inh} \bar{i} \mathbf{syn} \bar{s} z_2 \in \bar{v} \quad z_1 \sqsubseteq z_2 \quad c: I_c \bar{w} \in \Gamma_0 \quad \mathit{next} \bar{w} \bar{v} = x \quad \{\mathbf{inh} c: a \mid a \in \bar{i}\} \subseteq \Gamma_0 \quad \Gamma_1 = \Gamma_0 \cup \{\mathbf{syn} c: a \mid a \in \bar{s}\} \cup \{c: I_c(\bar{w}, \mathbf{visit} x \mathbf{inh} \bar{i} \mathbf{syn} \bar{s})\}}{\Sigma; \Gamma_0 \vdash \mathbf{invoke} x \text{ of } c z_1: \Gamma_1} \text{ INVOKE}$				
$\frac{\Gamma_0 \vdash e \quad \Gamma_1 = \Gamma_0 \cup \{c: I \emptyset\}}{\Sigma; \Gamma_0 \vdash \mathbf{child} c: I = e: \Gamma_1} \text{ CHILD} \quad \frac{\mathbf{inh} lhs: a \in \Gamma}{\Gamma \vdash lhs: a} \text{ OCC.LHS} \quad \frac{\mathbf{syn} c: a \in \Gamma}{\Gamma \vdash c: a} \text{ OCC.CHILD}$				
$\frac{\mathbf{syn} lhs: a \in \Sigma}{\Sigma; \Gamma_0 \vdash lhs: a: \Gamma_0, \mathbf{syn} lhs: a} \text{ PAT.LHS} \quad \Sigma; \Gamma_0 \vdash loc: a: \Gamma_0, \mathbf{syn} loc: a \text{ PAT.LOC}$				
$\frac{\mathbf{inh} c: a \in \Sigma}{\Sigma; \Gamma_0 \vdash c: a: \Gamma_0, \mathbf{inh} c: a} \text{ PAT.CHILD} \quad \Sigma; \Gamma \vdash K: \Gamma \text{ CONST} \quad \Sigma; \Gamma \vdash \_ : \Gamma \text{ ANY}$				
$\begin{aligned} \mathit{avail}(\mathbf{visit} x \bar{r} \bar{k}) &= \mathit{avail}_{\cup}(\bar{r}) \cup \mathit{avail}_{\cap}(\bar{k}) \\ &\quad \cup \{\mathbf{syn} lhs: b \mid \mathbf{visit} x \mathbf{inh} \bar{a} \mathbf{syn} \bar{b} \in \Phi(I_x)\} \\ \mathit{avail}(\mathbf{clause} x \bar{r} t) &= \mathit{avail}_{\cup}(\bar{r}) \cup \mathit{avail}(t) \\ \mathit{avail}(p = e) &= \emptyset \\ \mathit{avail}(\mathbf{match} p = e) &= \emptyset \\ \mathit{avail}(\mathbf{invoke} x \text{ of } c) &= \{\mathbf{inh} c: a \mid a \in \bar{a}, \mathbf{visit} x \mathbf{inh} \bar{a} \mathbf{syn} \bar{b} \in \Phi(I_c)\} \\ \mathit{avail}(\mathbf{child} c: I = e) &= \{c: I(\Phi I)\} \end{aligned}$				

Figure 3.9: Static semantics of RulerBack

### 3 AGs with Side Effects

represents an attribute, the attribute gets defined.

- **pin**?  $p = e$  -- eval-rule (optionally pinned)

Similar to the above, except that the match is expected to succeed. If not, the evaluation itself aborts with an exception. For that reason, we also call these rules assert-rules. Note that these are the conventional rules of attribute grammars.

- **child**  $c:I = e$  -- child-rule  
**child**  $root:Menu = ntMenu$  -- example that introduces a *Menu* child

Evaluation of the rule above creates a child  $c$ , visitable according to the interface  $I$ , and created by executing the constructor function  $e$ .

- **invoke**  $x$  of  $c$   $z$  -- invoke rule

Executes visit  $x$  of child  $c$ . The inherited attributes of  $x$  must be defined, and all prior visits to  $c$  must have been performed. The invocation fails if no clause matches and the strategy  $z$  is **partial**. Otherwise, the evaluation aborts exceptionally. If successful, the synthesized attributes of  $x$  become available. If there is an invoke with a partial-annotation, then the visit of the corresponding interface must also have a partial-annotation.

Figure 3.9 shows RulerBack’s static semantics for sem-blocks. We omitted the rules that state the uniqueness of interfaces and attributes of interfaces. A RulerBack program that satisfies these conditions never crashes due to an undefined attribute, invalid rule order, or forgotten invocation to a child. The dynamic or static type checking we leave as responsibility of the host language.

We briefly consider some aspects of these rules. Three environments play an important role. The environment  $\Phi$  contains for each interface the sequence of visits. The environment  $\Gamma$  represents the children and attributes defined so far for one node (to test for missing and duplicated definitions). The environment  $\Sigma$  represents the attributes that are allowed to be defined (to test for definitions of unknown attributes). As additional constraint on environments, we consider it a static error when there is a duplicate attribute in the environment within two scope markers.

Visit-blocks must be specified in the order as declared on the interface, and none may be omitted. The relation for visits  $t$  gets a sequence of pending visits  $\bar{v}$  as declared in the interface. In rule **VISIT**, we verify that the name of the visit matches the expected visit in the head of  $\bar{v}$ . The next visit must match the head of the tail of this list, until in the end  $\bar{v}$  is empty. We also add the inherited attributes of the visits to the environment.

The function *avail* defines which attributes may be defined. Higher-up in the visit-clauses-tree, we may only define those attributes that are common to all lower clauses. In rules **PAT.LHS** and **PAT.CHILD**, we verify that we are indeed defining an attribute belonging to a certain child. The *avail*-function either generalizes over lists using intersection or union.

Rule SEM forms the root of derivation trees. Since semantics blocks can occur nested,  $\Gamma$  contains potential bindings in scope from encapsulating semantics blocks. A new scope entry is added to  $\Gamma$ . The rule matches the visits blocks  $t$  against the declared visits  $v$ . Rule END matches with the end of a sequence of visits blocks, and requires also to have reached the end of the declared visits. Rule VISIT requires that the clauses and rules are well-formed. From the rules  $\bar{r}$  it obtains an environment  $\Gamma_1$  with additional bindings that are available to the clauses. Both the rules and the clauses may refer to the inherited attributes of the visit. Each clause must define the set  $\bar{s}$  of synthesized attributes of the visit. We assume that the type rule for a rule  $r$  is lifted to a list of rules  $\bar{r}$  by chaining the environment  $\Gamma$ .

Rule CLAUSE verifies that the rules it contains and the next visit are well-formed. Furthermore, it verifies that the synthesized attributes of the visit are defined. The type rules ASSERT, MATCH, INVOKE, and CHILD correspond to RulerBack rules. The ASSERT and MATCH rules verify that the pattern and expression are well-formed. The CHILD rule adds the child in an empty state to the environment.

Rule INVOKE verifies that visit  $x$  is indeed the next visit in the expected sequence of visits  $\bar{v}$ , given the previous invocations  $\bar{w}$ . It furthermore verifies that the inherited attributes for the visit of  $c$  are defined, and adds the synthesized attributes to the environment. Finally, the strategy annotation of the invoke-rule must be greater than the strategy annotation of the visit. If the visit is declared as total, then as sanity-check, an invoke may not have **partial** as strategy.

## 3.4 Translation of RulerBack to JavaScript

In this section, we describe how to translate RulerBack programs to JavaScript. We translate each semantics-block to a coroutine, which we implement as one-shot continuations. Each call to the coroutine represents a visit. The parameters of the coroutine are the inherited attributes of the visit. The result of the call is an object containing values for the synthesized attributes, and the continuation to call for the visit.

As an example, we show in Figure 3.10 the translation of the example in the previous section. To deal with backtracking, we use the exception mechanism, and throw an exception to switch to the next clause. Note that this does not rollback any side effects that the partial execution of the rules may have caused. To be able to do so, we can run the rules in a software transaction [Heidegger et al., 2010], which are nowadays supported by many programming languages. Alternatively, when the side effects matter, the programmer can schedule the rule to an earlier or later visit, such that it is not influenced by backtracking.

To deal with continuations, we use closures. The function to be used for the next visit is constructed in the previous visit. This function has access to all the results computed in the previous visit. Furthermore, we store values for attributes in local variables. Those values that are not needed anymore are automatically cleaned up by the garbage collector.

Figure 3.11 shows the general translation scheme, and the naming scheme for attributes. In particular, for each visit, we generate a closure that takes values for inherited attributes as parameter. Clauses are dealt with through exception handling. When a clause successfully executed all statements, it returns an object containing values for synthesized attributes, as

```

var sumArr = function () {
  function nt_sum (_inps) {
    var lhsII = _inps.l;
    try {
      if (lhsII.length != 0) throw eEval;
      var _res = new Object ();
      _res._next = function (_inps) {
        var lhsSs = 0;
        var _res = new Object ();
        _res._next = null;
        _res.s = lhsSs;
        return _res;
      };
      return _res;
    } catch (err) {
      var locLx = lhsII [0];
      var locLxs = lhsII.slice (1);
      var vis_tl = sumArr ();
      tIII = locLxs;
      var _args = new Object ();
      _args.l = tIII;
      var _res = vis_tl (_args);
      var vis_tl = _res._next;
      var _res = new Object ();
      _res._next = function (_inps) {
        var _args = new Object ();
        var _res = vis_tl (_args);
        var tISs = _res.s;
        var lhsSs = locLx + tISs;
        var _res = new Object ();
        _res._next = null;
        _res.s = lhsSs;
        return _res;
      };
      return _res;
    }
  };
}
return nt_sum;

```

-- semantic function  
 -- visit  $v_1$   
 -- extract  $lhs:l$   
 -- try clause  $sumNil$   
 -- if  $lhs:l$  is empty  
 -- produce results of  $v_1$   
 -- cont. for visit  $v_2$   
 --  $lhs:s$  rule  
 -- produce results of  $v_2$   
 -- no next visit  
 -- store  $lhs:s$   
 -- return result of  $v_2$   
 -- return result of  $v_1$   
 -- try clause  $sumCons$   
 --  $loc:x$  rule  
 --  $loc:xs$  rule  
 -- creation of child  $tl$   
 --  $tl:l$  rule  
 -- inputs for  $v_1$  of  $tl$   
 -- store  $tl:l$   
 -- invoke  $v_1$  of  $tl$   
 -- extract results  
 -- produce results of  $v_1$   
 -- cont. for visit  $v_2$   
 -- inputs for  $v_2$  of  $tl$   
 -- invoke  $v_2$  of  $tl$   
 -- extract  $tl:s$  result  
 -- compute  $lhs:s$   
 -- produce results of  $v_1$   
 -- no next visit  
 -- store  $lhs:s$   
 -- return result of  $v_2$   
 -- return result of  $v_1$   
 -- return visitor function

Figure 3.10: Example translation

well as the continuation function for the next visit.

The above translation is relatively straightforward. In practice, the selection of a clause is functionally dependent on the value of an inherited attribute, or a local attribute computed in a previous visit. In those cases, the selection of clauses can be implemented more efficiently using conventional branching mechanisms. Also, instead of using the exception mechanism to implement backtracking, we can use code duplication, which results in more efficient code. Chapter 7 shows such a translation scheme.

We verified that the above implementation runs in time linear in the size of the tree when we use a version of the *slice* operation that does not make a copy of the array. With a throughput of about hundred array elements per microsecond, and about a thousand per microsecond with the exception handling replaced by conventional branching, this is still about one or two orders of magnitude slower than using a hand-written loop. In our experience, however, the traversal performance is rarely an issue. In general, the asymptotic complexity of the traversal is linear in the size of the tree, and the actual time taken by traversing the trees is insignificant compared to the work performed by the right-hand sides of the rules in a real application.

## 3.5 Translation of RulerCore to RulerBack

In Section 3.2.4, we showed in an example how a RulerCore program can be encoded using only syntax of RulerBack. We omit the data-type driven translation from a **datasem** into a **sem**, nor the translation of default-rules. Instead, in this section we assume that RulerCore consists of those programs that after insertion of invoke-rules and reordering of rules are a valid RulerBack program.

### 3.5.1 Implicit Invocations

In RulerCore, invoke-rules may be omitted. From a RulerBack program, we derive a number of implicit invocations. We first determine the attributes that are *needed*. From these we determine the maximum needed visit, and thus the sequence of visits that is needed. An invoke-rule needs to be inserted if there is no invoke-rule for any of these visits yet. We start the insertion-process at the root of the tree, and check at each level downwards which invokes need to be inserted. With this process, we insert the invoke-rules at the lowest point, while still being in scope of all rules that need it. Automatic rule ordering then positions the invokes at their appropriate places.

A synthesized attribute  $a$  of child  $c$  is needed if there exists a rule which has the attribute reference  $c : a$  in its right-hand side. The needed attributes may differ per clause and visit, which we define in a similar way as *avail* in Section 3.3:

$$\begin{aligned} need(\mathbf{visit} \ x \ \bar{r} \ \bar{k}) &= need_{\cup} \ \bar{r} \cup need_{\cap} \ \bar{k} \\ need(\mathbf{clause} \ x \ \bar{r} \ t) &= need_{\cup} \ \bar{r} \cup need \ t \\ need(p = e) &= need \ e \end{aligned}$$

Note that *need* generalizes to lists by using either intersection or union, which we denoted explicitly with  $need_{\cup}$  and  $need_{\cap}$ .

$\llbracket \text{sem } x : I \ t \rrbracket$	$\rightsquigarrow$	<b>function</b> () { <b>var</b> $\llbracket nt \ x \rrbracket = \llbracket t \rrbracket_I$ ; <b>return</b> $\llbracket nt \ x \rrbracket$ ; }
$\llbracket c : x \rrbracket$	$\rightsquigarrow$	$\llbracket inp \ c \ x \rrbracket$
$\llbracket e \rrbracket_I$	$\rightsquigarrow$	<i>null</i>
$\llbracket \text{visit } x \ \bar{r} \ \bar{k} \ z \rrbracket_I$	$\rightsquigarrow$	<b>function</b> ( <i>_inps</i> ) { $\llbracket inp \ lhs \ (inhs \ I \ x) \rrbracket = \_inps.\llbracket inhs \ I \ x \rrbracket$ ; $\llbracket \bar{r} \rrbracket$ ; $\llbracket \bar{k} \rrbracket_{I,z,\text{syns } I \ x}$ ; }
$\llbracket [] \rrbracket_{I,\text{partial},\bar{s}}$	$\rightsquigarrow$	<b>throw</b> <i>eEval</i> ;
$\llbracket [] \rrbracket_{I,\text{total},\bar{s}}$	$\rightsquigarrow$	<b>throw</b> <i>eAbort</i> ;
$\llbracket k : \bar{k} \rrbracket_{I,z,\bar{s}}$	$\rightsquigarrow$	<b>try</b> { $\llbracket k \rrbracket_{I,\bar{s}}$ ; } <b>catch</b> ( <i>err</i> ) { <b>if</b> ( <i>err</i> == <i>eEval</i> ) { $\llbracket \bar{k} \rrbracket_{I,z,\bar{s}}$ ; } <b>else</b> <b>throw</b> <i>err</i> ; }
$\llbracket \text{clause } x \ \bar{r} \ t \rrbracket_{I,\bar{s}}$	$\rightsquigarrow$	$\llbracket \bar{r} \rrbracket$ ; <b>var</b> <i>_outs</i> = <i>new Object</i> (); <i>_outs</i> .. <i>next</i> = $\llbracket t \rrbracket_I$ ; <i>_outs</i> .. <i>s</i> = $\llbracket out \ lhs \ \bar{s} \rrbracket$ ; <b>return</b> <i>_outs</i> ;
$\llbracket \text{pin}^? \ p = e \rrbracket$	$\rightsquigarrow$	<b>var</b> <i>_res</i> = $\llbracket e \rrbracket_{\text{total}}$ ; $\llbracket p \rrbracket_{eAbort}$
$\llbracket \text{match } p = e \rrbracket$	$\rightsquigarrow$	<b>var</b> <i>_res</i> = $\llbracket e \rrbracket$ ; $\llbracket p \rrbracket_{eEval}$ ;
$\llbracket \text{child } c : I = e \rrbracket$	$\rightsquigarrow$	<b>var</b> $\llbracket vis \ c \rrbracket = (\llbracket e \rrbracket) ()$ ;
$\llbracket \text{invoke } x \ \text{of } c \ z \rrbracket$	$\rightsquigarrow$	<b>var</b> <i>_args</i> = <i>new Object</i> (); <i>_args</i> .. $\llbracket inhs \ I_c \ x \rrbracket = \llbracket out \ c \ (inhs \ I_c \ x) \rrbracket$ ; <b>var</b> <i>_res</i> = $\llbracket vis \ c \rrbracket (\_args)_z$ ; <b>var</b> $\llbracket inp \ c \ (syns \ I_c \ x) \rrbracket = \_res.\llbracket syns \ I_c \ x \rrbracket$ ; <b>var</b> $\llbracket vis \ c \rrbracket = \_res..\text{next}$ ;
$\llbracket e \rrbracket_{\text{partial}}$	$\rightsquigarrow$	<i>e</i>
$\llbracket e \rrbracket_{\text{total}}$	$\rightsquigarrow$	<b>try</b> { <i>e</i> ; } <b>catch</b> ( <i>err</i> ) { <b>if</b> ( <i>err</i> == <i>eEval</i> ) <b>throw</b> <i>eAbort</i> ; <b>else</b> <b>throw</b> <i>err</i> ; }
$\llbracket c : a \rrbracket_e$	$\rightsquigarrow$	<b>var</b> $\llbracket out \ c \ a \rrbracket = \_res$ ;
$\llbracket - \rrbracket_e$	$\rightsquigarrow$	;
$\llbracket k \rrbracket_e$	$\rightsquigarrow$	<b>if</b> ( <i>_res</i> != <i>k</i> ) <b>throw</b> <i>e</i> ;

$out \ "loc" \ x = "locL" \ x$      $inp \ "loc" \ x = "locI" \ x$   
 $out \ "lhs" \ x = "lhsS" \ x$      $inp \ "lhs" \ x = "lhsI" \ x$   
 $out \ c \quad x = c \ "I" \ x$      $inp \ c \quad x = c \ "S" \ x$   
 $vis \ c = "vis\_ " \ c$          $nt \ x = "nt\_ " \ x$   
 $syns \ I \ x \quad inhs \ I \ x$     -- respectively, inh and syn attrs of *x* of *I*

Figure 3.11: Denotational semantics of RulerBack



In our actual implementation, we defined the function *need* in a slightly more subtle way. A default-rule may indirectly express a need on an attribute (and corresponding visit). Furthermore, when a programmer provided an explicit invoke rule for a visit of a child *c*, then the programmer must give explicit invoke rules in all clauses that require attributes of this visit of *c*. This is a policy that we impose, because apparently the programmer had a reason to explicitly specify an invocation of the visit, instead of using the implicit specification.

### 3.5.2 Rule Ordering

To order the rules, we first create production dependency graphs (PDGs) (Section 1.3.4). Note that we do not need nonterminal dependency graphs, because these are fully implied by the interface of a nonterminal. In comparison to conventional PDGs, the PDGs of RulerCore contain also vertices that represent visits, clauses, and invocations of visits. The graph is also slightly less complex because rules do not depend on synthesized attributes of children, but on the visits of the children that produce these attributes.

In the PDG of a production (thus, a semantics block), there exists a begin and end vertex for each visit in the interface. There exists also a vertex for each clause and each rule. Figure 3.12 lists the dependencies between vertices, and gives an impression of the dependency graph of *Menu* in Figure 3.3. The dependency graph is acyclic, thus the rules can be ordered.

In this sketch, the ovals represent rules, and the square boxes represent the begin and end of visits and clauses. The numbers represent the line numbers<sup>3</sup>. The squares immediately following the root are the begin points of the clauses for that visit. Clauses are constrained by begin and end points of visits. Therefore, branches come together again. Some rules are constrained to visits (notably match rules). For other rules, we have more flexibility in their scheduling. For *Menu*, we did not define clauses for the second visits, hence the implicit clause in the graph.

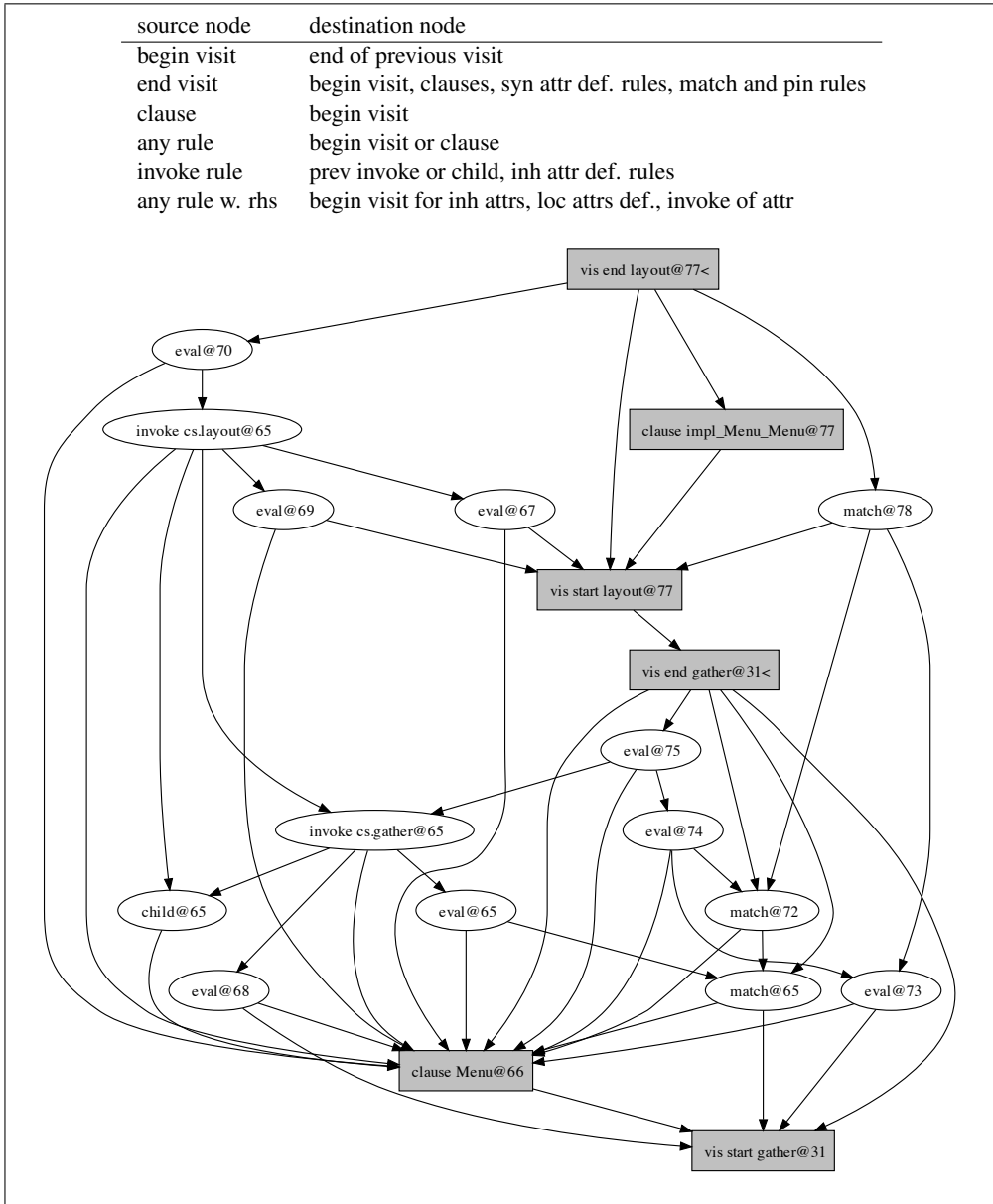
Note that we consider the edges in the direction of their dependency, not in the direction of the value flow. Thus, *p* is a successor of *q* if *p* needs to be defined before *q*. To make the distinction clearer, we refer to the direct and indirect successors as *dependencies* and direct and indirect predecessors as *users*.

The acyclic graph represents a partial order between rules, visits and clauses. We turn the partial order in a total order in a number of preprocessing steps. Any total order that satisfies the partial order must result in a semantically equivalent result. However, differences between total orders may affect the performance of the resulting algorithm. In the dependency graph, each rule is associated with a unique vertex.

**Step 1: order by visit.** First, we determine for each rule to which visit to schedule it. Sometimes, rules can be scheduled to more than one visit. The visit a rule is scheduled to influences the amount of data that has to be transported between visits. The inputs for the rule need to be transported to the visit of the rule, and the result of the rule to the visits where these results are used. The set of visits to which a rule *r* of a production of nonterminal *N*

<sup>3</sup> See <https://svn.science.uu.nl/repos/project.ruler.systems/ruler-core/examples/JsMenu.rul>.

### 3 AGs with Side Effects



**Figure 3.12:** Dependencies between RulerCore entities, and the dependency graph of *Menu*.

can be scheduled to are all visits of  $N$ , except the visits associated with end-visit nodes in the dependencies of  $r$ , and the visits associated with the begin-visit nodes in the users of  $r$ .

Given these sets of visits for rules, we can apply scheduling strategies. For example, in Chapter 5, we present notation to specify the iteration of visits. We try to avoid scheduling a rule to such visits, if possible.

We may schedule rule  $r$  to any of these visits, however, by doing so, the scheduling may affect the set of possible visits for dependencies and users of  $r$ . As our default scheduling strategy, we schedule each rule to its last possible visit<sup>4</sup>, and add correspondingly a scheduling dependency from the rule to the beginning of that visit to the graph. This approach has the advantage that the order in which we make decisions for rules does not influence the resulting graph, and also ensures that the graph remains acyclic.

**Step 2: order by partition.** Secondly, we determine per visit the order of its rules. The order of the rules may affect the amount of overhead in the clause selection. We need to consider the following items.

- Rules are preferably executed once per visit. As heuristic, we schedule rules that do not depend on any clause before the clause selection<sup>5</sup>.
- Match-rules and invoke-rules with the partial-strategy may fail. Such rules are preferably scheduled early, because all computations that are performed for a clause that fails is overhead. For example, the match-rules that test the value of an inherited attribute, as introduced by the translation of **datasems**, should be scheduled upfront.
- Rules may have expressions that are expensive to compute. An accurate estimation of such costs requires an analysis of host-language terms and is hard in general.
- Rules of a particular form may depend on other rules of a different form. We therefore cannot straightforwardly group all match-rules together.

To take these items into account, we use a customizable strategy. We associate with each rule a *partition* and *class* for a finite sequence of partitions and a finite sequence of classes. Rules of one partition are scheduled before rules of a later partition. Within a partition, we schedule rules iteratively. With each iteration, we schedule the rules of the earliest class that has rules that can be scheduled. With this approach, rules of an earlier class precede rules of a later class, if possible. We describe this approach in more detail below.

We define an ordered sequence of partitions. Each rule must be associated with one partition, and a rule may not have a dependency that belongs to a later partition. For a given visit, we schedule the rules of one partition before those of the next partition. The default partitions are:

---

<sup>4</sup> Each nonterminal has a terminator visit  $\epsilon$ , which does not have any attributes, and for which no code is generated. Rules that are scheduled to that visit are discarded during code generation.

<sup>5</sup> An alternative approach is to keep track of which rules that do not depend on clauses are already applied during the evaluation of a previous clause. However, the savings on overhead are likely small. For visits declared as total, such rules eventually need to be applied.

### 3 AGs with Side Effects

**data**  $P = InCommon \mid InClause$  **deriving** ( $Eq, Ord$ )

We associate with *InCommon* all rules of a visit that are not users of a begin-clause-node of that visit, and with *InClause* the remaining rules of that visit.

type of rule	classification	strategy
match-rules	<i>Earliest</i>	quick clause selection
partial invoke-rules	<i>Earlier</i>	clause selection
pin-rules	<i>Middle</i>	early side effect, preferably no backtrack
total invoke-rules	<i>Later</i>	visit children when needed
eval-rules	<i>Latest</i>	minimize distance between computation and use

**Figure 3.13:** Association of rules with classes.

**Step 3: order by classification.** For each partition, we define the order of the rules based on a classification of the rule. Each rule must be associated with a class, which represents a scheduling preference. By default, we distinguish the following classes:

**data**  $C = Earliest \mid Earlier \mid Middle \mid Later \mid Latest$  **deriving** ( $Eq, Ord$ )

Figure 3.13 shows the default association based on a rule's type. To influence the ordering, a programmer can specify more classes, and explicitly specify such a class for some of the rules.

The classes specify a scheduling preference: a rule of an earlier class is preferably scheduled before a rule of a higher class. However, this is not possible when a rule of an earlier class depends on a rule of a later class. Moreover, to schedule some rules of an earlier class, there may be different *minimal* subsets of rules of a later class possible, so that it is not obvious which subset to take.

In the following example, the first match-rule is scheduled first. None of the other match-rules can be scheduled, unless either the rule for *loc.a* or the rule for *loc.b* is scheduled before. To ensure a deterministic scheduling, we schedule both *loc.a* and *loc.b* rules, and use their order of appearance as final distinguishing factor:

```

match loc.x = 3      -- class: Earliest
loc.a = e           -- class: Latest
loc.b = loc.x       -- class: Latest
match loc.y = f loc.a -- class: Earliest
match loc.z = g loc.b -- class: Earliest

```

However, if the rule for *loc.y* would additionally mention *loc.z* in its RHS, then the rule for *loc.a* is scheduled directly after the rule for *loc.x*, followed by the rule for *loc.z*, and only then the rule for *loc.b*.

Figure 3.14 shows the scheduling algorithm *rankVertices*. It uses the rank monad *R*, which is a combination of a list, state, and continuation monad with monad comprehensions [Middelkoop, 2011c], to describe algorithms with the enumeration combinator *foreach* and the

```

-- gives each vertex of verts a unique rank based on classes
rankVertices :: [C] → [Node] → R ()
rankVertices classes verts = do
  c ← foreach classes
  iter $ do
    vs ← unionM [ps | v ← verts, hasClass v c, let ps = deps' verts v
                  , allM (λn → hasClass n c 'impliesM' isRanked n) (ps \\ [v])]
    guard (notNull vs)
    iter $ do
      vss ← filterM notNull $ mapM (readyNodes verts vs) classes
      guard (notNull vss)
      v ← foreach $ sortAsc cmpNodePos $ head vss
      rank v

-- returns the subset of vs that can be scheduled
readyNodes :: [Node] → [Node] → C → R [Node]
readyNodes verts vs c =
  [v | v ← vs, isNotRanked v, hasClass v c, allM isRanked (deps verts v)]

-- returns v and its indirect dependencies
deps' :: [Node] → Node → [Node]
deps' verts v = [v] ∪ map (deps' verts) (deps verts v)

iter      :: R () → R ()           -- iterate param until a guard is False
guard    :: Bool → R ()          -- check a condition; fail if False
foreach  :: [a] → R a            -- repeat cont. for each item in list
rank     :: Node → R ()          -- gives the node the next rank
deps     :: [Node] → Node → [Node] -- direct dependencies
hasClass :: Node → C → R Bool   -- True iff the node has the given class
isRanked :: Node → R Bool       -- True iff the node is ranked
sortAsc  :: (Node → Node → Ordering) → [Node] → R [Node]
cmpNodePos :: Node → Node → Ordering

```

Figure 3.14: Order algorithm for the visit's rules.

### 3 AGs with Side Effects

iteration combinator *iter*. As parameters, it gets the sequence of classes in the order from earliest to latest, and the set of vertices of the partition to schedule. The result of the algorithm is an association with a unique rank for each of these vertices.

With this algorithm, we try to schedule rules in increasing class order by giving the vertices of such rules an increasing rank. The class  $c$  is the class we currently schedule for. For class  $c$ , we determine the set  $vs$ , which is the smallest set that contains all vertices of rules of class  $c$  that do not depend on an unranked vertex of class  $c$  or an earlier class. Moreover, the set contains the indirect dependencies of these rules. This set may include vertices of a later class, or already ranked vertices. Thus,  $vs$  contains the largest set of vertices of class  $c$  that can be ranked when we only consider already ranked vertices of the same class or earlier. Also, it contains the smallest set of vertices of a later class that need to be ranked to allow the ranking of all the vertices of an earlier class in  $vs$ . For an acyclic graph, and as long as there are unscheduled rules of class  $c$ , the set  $vs$  has at least one rule. We ensure that each rule in  $vs$  is scheduled. Thus, by repeating this process we rank all rules of class  $c$ . Consequently, after processing all classes, all vertices in *verts* are ranked.

We rank the vertices of  $vs$  in iterations. With *readyNodes* we determine for each class the set of unranked vertices that have ranked dependencies, and take the nonempty subset of the earliest class. Such a set exists, unless all vertices of  $vs$  are ranked. We rank these vertices in the order of the appearance of their rules. The algorithm guarantees that an unranked node is ranked if all its dependencies are ranked (soundness property), and that there is per iteration at least one vertex ranked as long as there are unranked vertices left (progress property).

Assume the number of vertices to be ranked is  $n$ , and the number of classes is  $c$ . Both the worst-case asymptotic time and memory complexity of *deps'* is  $O(n)$ , where  $|vs| \leq n$ . The functional implementation assumes that vertices are represented as an integer. The corresponding rule can be determined in constant time via a lookup table. For *readyNodes* the time and memory complexities are  $O(n^2)$ . The time complexity of the algorithm for  $vs$  is thus  $O(cn^2)$ , and of the algorithm for  $vs$  is  $O(n^2)$ . The memory complexity is  $O(n^2)$ . Despite the double iter-blocks, each line is at most repeated  $n + c$  times, because with every repetition at least one vertex is ranked. The worst-case time complexity of *rankVertices* is thus  $O(cn^3)$  and the worst-case memory complexity is  $O(n^2)$ . Note that in practice the values for  $c$  and  $n$  are small, and that most rules have only a small number of dependencies.

**Step 4: cleanup.** We schedule the rules in a partition in the order of their rank. For the purely functional eval-rules, we make an exception. Starting with the highest-ranked eval-rule, we shift it just before the lowest-ranked of its users in the dependency graph. Scheduling such rules later does not affect the overhead of clause selection, and our strategy is to bring such rules close to where their results are used. Since eval-rules have the *Latest*-classification, it is already likely that these rules are already at such a position. In a similar way, we could also move total invoke-rules. However, the visit invoked by total invoke-rules may contain pin-rules, thus for such rules we keep the scheduling via the classification mechanism.

**Remarks.** The implicit invokes and automatic ordering allow a straightforward transformation from a dataseM-block to a sem-block. Essentially, a dataseM-block is syntactic sugar

for a number of clauses, each with a match-rule, and a number of child-rules. We also allow in RulerCore omitted visits and clauses to be implicitly defined. Combined with implicit invocations, this makes it easy to add additional visits to an interface. Furthermore, the automatic rule ordering allows us to write independent rules separately from each other (possibly in separate files) and use a preprocessing step to merge the rules together.

The scheduling also offers opportunities to exploit parallelism. When the *head vss* consists of total invoke-rules, during *rankVertices*, these invocations are candidates to evaluate in parallel because their computations are independent. However, whether or not parallelism is beneficial depends on how expensive the computation of the visit is. With the classification mechanism, a rough static approximation can be expressed by the programmer. For example, we can introduce a class for total invocation rules that take priority over conventional invocation rules. When multiple of these invocation-rules are scheduled together, we can actually generate code that performs the visits in parallel.

## 3.6 Discussion

RulerCore can be used to express traversals over tree-like data structures. To a limited extent RulerCore may be applicable to graphs traversals that are technically tree traversals (such as a traversal over a depth-first forest). Loops and iteration can be expressed with higher-order attributes. In related work, we expressed these by iterating visits [Middelkoop et al., 2010a]). However, RulerCore is not suitable to express traversals over drastically changing data structures.

In our actual implementation, we also provide a notion of internal visits. A conventional visit is invoked externally by the parent, and can choose a clause. This means that we can only conditionally compute attributes once per visit. In contrast, an internal visit is invoked at the end of the clause, and is not visible externally. An internal visit may again have clauses, and these clauses may again specify an internal visit as next visit, or a conventional visit. With this relatively simple extension, we can arbitrarily often branch inside a visit.

In the Haskell version of RulerCore, we require type signatures for attributes. In JavaScript, instead of type signatures, the notion of a type signature represents a dynamic check in the form of assertion-functions that validate the values for attributes.

To fully enjoy the benefits of attribute grammars, the host language requires support for purely functional data structures [Okasaki, 1998]. Such data structures can be encoded in JavaScript, but efficient versions with copy-on-write semantics are cumbersome to implement manually.

## 3.7 Related Work

Related to this chapter are various visitor-like approaches and attribute grammar techniques.

The purpose of the Visitor design pattern [Gamma et al., 1993] is to decouple traversal operations from the specification of the tree to be traversed, in order to make it easier to add new operations without changing the existing specification of the tree. This allows us to write a multi-visit traversal using a separate visitor per traversal. Multi-methods [Chambers

and Leavens, 1994] are supposed to replace the visitor pattern. A multi-method allows overloading of methods on multiple parameters, and makes *accept*-methods superfluous. This, however, is orthogonal to the problems and solutions that we presented in this chapter.

In Section 3.2.1, we discussed advantages and disadvantages of modeling traversals with visitors. In particular, side effects are permitted, and used to store results for use in later visits. The side effects make it hard to predict if results needed in a next visit are actually stored by a first visit. This is a fundamental problem of visitors. Oliveira et al. [2008], for example, show many enhancements with respect to the type safety of visitors, but do not address the transfer of results between visits.

Attribute grammars [Knuth, 1968, 1990] are considered to be a promising implementation for compiler construction. Several attribute grammar techniques are important for our work. Kastens [1980] introduces ordered attribute grammars. In OAGs, the evaluation order of attribute computations as well as attribute lifetime can be determined statically, allowing severe optimizations.

Boyland [1996] introduces conditional attribute grammars. In such a grammar, semantic rules may be guarded. A rule may be evaluated if its guard is satisfied. Evaluation of guards may influence the evaluation order, which makes the evaluation less predictable. In comparison, in our clauses-in-visits model, we have to explicitly indicate in which visits guards are evaluated (the match-statements of a clause), which makes it clear what the evaluation order is. Our approach has the additional benefit that children may be conditionally introduced and visited.

Recently, many Attribute Grammar systems arose for mainstream languages, such as Silver [Van Wyk et al., 2008] and JastAdd [Ekman and Hedin, 2007] for Java, and UUAG [Löh et al., 1998] for Haskell. In contrast to the work in this chapter, these systems strictly discourage or disallow the use of side effects. The design of RulerBack is inspired by the language of execution plans of UUAG. In certain languages, AGs can be implemented via meta-programming facilities, which obviates the need of a preprocessor. Viera et al. [2009] show how to implement AGs in Haskell through type level programming. The ideas that we presented in this chapter are orthogonal to such approaches, although the necessary dependency analysis may be difficult to express depending on the expressiveness of the meta language.

## 3.8 Conclusion

We introduced the language RulerCore, an extension of Attribute Grammars that makes visits to nonterminals explicit. As a consequence, it is possible to use side effects in rules. RulerCore combines the freedom of visitors as described by the visitor design pattern with the convenience of programming with attributes, as shown in Section 3.2.

Moreover, we presented RulerBack, a subset of RulerCore, which serves as a small core language for visitor-based Attribute Grammars. In RulerBack, the lifetime of attributes is explicit, as well as the evaluation order of rules and visits to children. We described how RulerCore programs are mapped to RulerBack in Section 3.5. A RulerBack program has a straightforward translation to many languages. In Section 3.4, we gave a translation to



JavaScript by making use of exceptions in combination with the try-catch mechanism. A more sophisticated translation is possible that does not require exceptions, as we show in later chapters.

**Future work.** A direction of future work is to consider destructive updates on attributed trees. Event-handling traversals over data structures may need to respond to dynamic changes induced by user input or external events. In RulerFront, the visits performed on an attributed tree explicitly specify which attributes are defined. When we apply a destructive update to the tree, we thus know precisely what information is based upon the previous structure of the tree. This knowledge can be exploited to reason about mutations of the attributes tree. Incremental evaluation of attribute grammars [Vogt et al., 1991, Yeh and Kastens, 1988] may be used to efficiently recompute attributes after modifications of the AST.

This chapter treated the scheduling of rules in the presence of side effects. Side effects are not visible in value dependencies between attributes. In Chapter 7 we show how to incorporate dependently-typed programming languages, which have type assumptions resulting from pattern matches as ‘effect’.



## 4 AGs with Commuting Rules

In Chapter 3 we described a programming model based on visits as an extension of attribute grammars. Using this model, attribute declarations are not as easily composed as with conventional attribute grammars. In this chapter, we introduce phases as a generalization of visits, for which attribute declarations are composable.

Further, we introduce commuting rules, which are rules that thread a chained attribute (Section 1.3.11 and Section 1.3.12), with the difference that the rules in the chain can be reordered. To preserve referential transparency, the commuting rules must satisfy a liberal commutativity law.

This chapter is orthogonal with respect to the subsequent chapters. It presents the technical material which can be used to generalize the contents of the subsequent chapters.

### 4.1 Introduction

In this chapter, we show three related extensions of the explicit visit-approach in Chapter 3 and ordered attribute grammars in general.

**Definition** (Phase). A phase is a generalization of a visit. It gives a name to a unit of evaluation for attributes associated with a nonterminal symbol.

Firstly, instead of associated visit declarations with nonterminals as in the previous chapter, we associate sets of phase declarations with nonterminals in this chapter. We allow attribute declarations and rules to depend on the beginning of a phase or be constrained by the end of a phase, and impose a partial order on phases. When the value dependencies between attributes and rules and the dependencies induced by phases are acyclic, the attributes that are associated to a previous phase are defined before any attributes that are associated with a later phase, which provides a way to specify the order of evaluation of rules.

Secondly, we use a statically ordered evaluation to compute values of attributes, and we present a scheduling algorithm in the style of Kennedy and Warren [1976] which infers multiple visit interfaces from the phase interface of a nonterminal. For each visit interface, a production has a potentially different execution plan. An execution plan specifies an explicit ordering of rules per visit (Section 1.3.3). Such a plan specifies also for each child which visit interface is used for its evaluation.

In the generated code, semantic functions are indexed by the choice of the visit interface for which it provides a semantics, and we show how to encode this in a strongly typed, purely functional language. There may be many possible visit interfaces given the phase interface of a nonterminal. The constraints on rule and attribute ordering as mentioned above can be used to significantly limit the number of possible visit interfaces.

## 4 AGs with Commuting Rules

Finally, we present commutable rules, which allows us to functionally encode side effects. A functional encoding of side effects as mentioned in Chapter 3 can be accomplished with a chained attribute that is threaded through each operation, where an operation is a rule or higher-order child. For example, a substitution attribute captures the side effects that result from substitutions during type inference, and an attribute with the type *RealWorld* captures arbitrary I/O. The attribute's threading determines the order in which the side effects are observable in the values of the attribute.

For some threaded attributes, the order imposed by the threading may be irrelevant, for example when the rules represent *commuting* operations, such as when handing out unique numbers. However, the order imposed by threading the attribute may induce cycles in the dependencies of attributes. We present a mechanism to allow the order to be left unspecified and given the freedom to be determined by the implementation.

**Definition** (Commutable rule). A *commutable rule* is a rule that threads a chained attribute, but does not depend (indirectly) on previous rules in the chain.

Commutable rules thus provide more freedom in their ordering. To preserve referential transparency, the composition of commutable rules must satisfy a liberal commutativity law, as we show later. Commutable rules also allow the functional encoding of a side-effectful rule that is scheduled to different implicit visits.

We thus offer a mechanism to explicitly enforce constraints on the evaluation order of attributes, and another mechanism to loosen the constraints imposed by rules. The combination offers convenient ways to declaratively specify side-effectful operations and their algorithmic evaluation order.

Phases and commutable rules have in common that the associated scheduling algorithm takes information into account that is not visible in the attribute dependencies induced by the rules. Phases induce ordering constraints based on dependencies between attributes and the lexical scope of rules. Commutable rules require a barrier between rules that commute over an attribute and rules that do not. Therefore, we introduce barrier attributes and dependency rules, which allow the encoding of such additional dependencies.

**Definition** (Designator). A *designator*  $d$  (Figure 4.2) is either an attribute occurrence or a symbolic representation of a rule.

**Definition** (Dependency rule). A *dependency rule*  $d_1 \prec d_2$  is a rule that specifies that designator  $d_1$  must be scheduled before designator  $d_2$ .

**Definition** (Barrier attribute). A *barrier attribute* is an attribute that can be used as a *designator* (dependee/depender) in combination with dependency rules. The value of a barrier attribute is implicitly defined.

In this chapter, we first work out the concepts of barrier attributes and dependency rules (Section 4.2), and then show how these concepts can be exploited to deal with phases (Section 4.8) and commutable rules (Section 4.9).

```

grammar Tree                                     -- grammar for nonterm Tree
  | Leaf x :: Int                               -- production Leaf with one terminal
  | Bin l,r :: Tree                             -- production Bin with two nonterminals
attr Tree   syn gath :: Int                 -- attributes for nonterm Tree
                inh mini :: Int
                syn repl :: Tree
sem Tree                                       -- semantics for nonterm Tree
  | Leaf lhs.gath = loc.x
                lhs.repl = Leaf lhs.mini       -- replacement tree
  | Bin lhs.gath = min l.gath r.gath           -- global minimum gathering
                l.mini   = r.gath              -- crossover between r and l
                r.mini   = l.gath
                lhs.repl = Bin l.repl r.repl   -- replacement tree

```

Figure 4.1: A variant of the Repmin example.

## 4.2 Example with Barriers

We use Haskell as host language. Figure 4.1 shows a variant of the classical Repmin example [Bird, 1984], which requires more than one visit to compute the attributes with a statically ordered evaluation strategy. The attribute *repl* contains a clone of the tree with each leaf-child replaced with the minimum of its sibling's subtree. In order to get the *min* attribute of *l*, the *gath* attribute of *r* is needed and vice versa. With statically ordered evaluation, this can be accomplished by first computing the *gath* attribute in a first visit, then compute *min* and *repl* in a later visit.

**Barriers.** The evaluations for *l.repl* and *r.repl* are independent. For debugging purposes, we may want to specify that the *l.repl* attribute is computed before the *r.repl* attribute. Since the *syn.repl* attribute depends on the *inh.min* attribute, we get the desired behavior when *l.repl* is a dependency of *r.min*. The following dependency rule expresses this dependency:

```

sem Tree | Bin   order l.repl <- r.min

```

This rule requires that *inh.min* is a dependency of *syn.repl*. Note that, as usual, the dependency relation is transitive.

The dependencies imposed by dependency rules may not be clear when multiple synthesized attributes are involved, nor remain stable after code changes. For this purpose, we introduce a barrier attribute:

```

attr Tree   inh sync barrier
sem Tree

```

#### 4 AGs with Commuting Rules

```

| Leaf  order lhs.sync < lhs.repl
| Bin   order lhs.sync < l.sync
        order lhs.sync < r.sync
        order l.repl  < r.sync  -- actually subsumes lhs.sync < r.sync

```

Barrier attributes are not defined by a rule. Instead, these attributes are defined implicitly. However, we require that for a synthesized barrier attribute  $y$ , that there is at least one production with a dependency rule where  $lhs.y$  occurs as RHS, and at least one production with a child  $k$  where  $k.y$  occurs as LHS in a dependency rule. For an inherited barrier attribute  $y$ , we require the inverse. There is actually no technical reason for this requirement: if the requirement is not met, it should be taken as a warning that a barrier is not used.

**Definition** (Updatable attribute). An updatable attribute is an attribute that has a value which is a reference into a shared state. Via attribute occurrences (Figure 4.2), we specify operations on this shared state. An attribute occurrence  $z^\circ$  in expressions represents the value in the shared state referenced by  $z$ . An occurrence  $z^\circ$  in a pattern means that the shared state at the location pointed to by  $z$  is updated with the matched value. An occurrence  $z^\times$  in a pattern means that a new shared location is allocated with the matched value and a reference to it stored in  $z$ .

We incorporate side-effect in BarrierAG (Chapter 3) in the form of updatable attributes. In the following example an updatable attribute  $unq$  is created, read from, and updated:

```

attr Tree  inh unq :: IORef Int  -- inherited unique number dispenser
sem Root | Root
    root.unq× = 1                -- creates reference and assigns initial value
sem Tree | Leaf                -- reads and updates reference
    (loc.myId, alhs.unq°) = (alhs.unq°, alhs.unq° + 1)

```

Normally, attribute occurrences in a pattern are at defining positions. However, when  $z^\circ$  is used to specify a store to a shared state, the attribute occurrence  $z$  is at a usage position. Hence,  $alhs.unq$  refers to the inherited attribute  $alhs.unq$ , and is a dependency of the rule.

Rules that write to a reference are guaranteed to be applied at most once. The order in which these writes take place depends on the scheduling of attributes. If the attribute  $loc.myId$  is not used, the write actually does not take place. The only guarantee that is given for such a write is that the reference is created and initialized. For more guarantees, barriers can be used to control and specify the order.

**Remarks.** Barriers and updatable references should be used sparingly. In particular, rules with updatable references are not functional, which breaks equational reasoning. We present them here as an implementation mechanism for phases and commutable rules. With the later property, we can recover equational reasoning. Note that the code generated from an AG may be functional even if the AG description itself is non-functional.

$I ::= \mathbf{attr} N \bar{a}$	-- attr decls for nonterminal $N$
$a ::= k x t$	-- attribute declaration with form $k$ , attr name $x$ and type $t$
$k ::= \mathbf{inh} \mid \mathbf{syn}$	-- attribute forms (often also written as identifier)
$t ::= :: \tau$	-- attribute type (host language)
<b>barrier</b>	-- barrier type
$s ::= \mathbf{sem}_{\Gamma} P : N \bar{r}$	-- semantics of a production $P$ of nonterm $N$ in env $\Gamma$
$r ::= \mathbf{child} x : N = f \bar{z}^{\triangleright}$	-- (higher order) child declaration
$x : p \bar{z}^{\triangleleft} = f \bar{z}^{\triangleright}$	-- evaluation rule named $x$ with LHS $p$ and RHS $f$
<b>order</b> $d_1 \prec d_2$	-- order declaration
$d ::= \mathbf{child} x$	-- designates child $x$
<b>rule</b> $x$	-- designates a rule named $x$
$z$	-- designates an attribute occurrence
$z^{\triangleleft} ::= z^{\bullet}$	-- store in occurrence $z$
$z^{\circ}$	-- write to location referenced by $z$
$z^{\times}$	-- store new handle in $z$
$z^{\triangleright} ::= z^{\bullet} \mid z^{\circ}$	-- respectively read (closed), and deref (open)
$z ::= h.c.x$	-- occurrence attr $x$ of $c$ with form $h$
$h ::= k \mid loc$	-- attribute forms (extended with locals)
$c ::= lhs \mid loc \mid x$	-- child designators
$x, f, p, P$	-- identifiers

Figure 4.2: AG core representation.

### 4.3 Core Representation of AGs with Barriers

In this section, we define the semantics of the dependency rules and barrier attributes using the core language BarrierAG, a desugared subset of higher-order AGs. BarrierAG also permits attributes that have references to a global state as value, which we need later.

**Syntax.** Figure 4.2 gives the syntax of BarrierAG. BarrierAG serves as a core language for AGs. Hence, we assume that static checks, desugaring, item grouping, and copy rule insertion have been performed (Section 1.3.6 and Section 1.3.12). This core representation serves two purposes: it allows formal reasoning with AGs, and specifying the construction of dependency graphs and execution plans (also see Section 1.3.2).

The main (meta) nonterminals in the (meta) grammar of Figure 4.2 are  $I$  (attr-block) and  $s$  (semantics blocks). An attr-block consists of a set of attribute declarations. A semantics block provides the rules for a single production. The environment  $\Gamma$  is often left implicit.

Figure 4.3 shows a desugared version of the earlier example in BarrierAG. The context free grammar is translated to terms in the host language, and is not part of the core language. The symbols of the production are represented as higher-order children and local attributes.

```

sem_Tree (Leaf x) = sem_Leaf x
sem_Tree (Bin l r) = sem_Bin (sem_Tree l) (sem_Tree r)
attr Tree   syn  gath :: Int
             inh  mini :: Int
             inh  sync barrier
             syn  repl :: Tree

sem_Leaf field_x =
  sem {f1 = field_x, f2 = Leaf} Leaf : Tree
  r0: id loc.loc.x• = f1
  r1: id syn.lhs.gath• = id loc.loc.x•
  r2: id syn.lhs.repl• = f2 inh.lhs.mini•
  order inh.lhs.sync < syn.lhs.repl

sem_Bin field_l field_r =
  sem {f1 = field_l, f2 = field_r, f3 = min, f4 = Bin} Bin : Tree
  child l : Tree = f1
  child r : Tree = f2
  r3: id syn.lhs.gath• = f3 syn.l.gath• syn.r.gath•
  r4: id syn.lhs.repl• = f4 syn.l.repl• syn.r.repl•
  r5: id inh.l.mini• = id syn.r.gath•
  r6: id inh.r.mini• = id syn.l.gath•
  order inh.lhs.sync < inh.l.sync
  order inh.lhs.sync < inh.r.sync
  order syn.l.repl < inh.r.sync
ast = sem_Bin (sem_Leaf 1) (sem_Leaf 2)

```

Figure 4.3: Desugared example.

Evaluation rules are explicitly named, and consist of a function symbol  $p$  and  $f$ . The evaluation rule represents the function  $\bar{z}_1 = p(f \bar{z}_2)$ . For example, the rule  $r_3$  represents the function  $\text{syn.lhs.gath} = \text{id}(f_3 \text{syn.l.gath} \text{syn.r.gath})$ . The definition of these functions are bound in the environment  $\Gamma$ , which is an annotation of the sem-block. This representation has the advantage that the rules can be duplicated without duplicating the body of the function.

**Semantics.** Ultimately, we generate host-language code for the core language term. However, to facilitate reasoning with terms in the core language, we first define an operational semantics<sup>1</sup> in Figure 4.4 which denotes a tree walking automaton (Section 1.3.3). The semantics refers to rules in Figure 4.5, which can be interpreted as a dynamic version of the

<sup>1</sup> This operational semantics is also implemented as part of the `ruler-interpreter`: <https://svn.science.uu.nl/repos/project.ruler.papers/archive/ruler-interpreter-0.1.tar.gz>.



dependency analysis that we define in the next section or as a specification of a tree-walking automaton (Section 1.3.3). Since we deal with higher-order AGs, the operational semantics describes both the construction and decoration of the AST.

A decorated node of the AST is represented as a tuple  $(N, P, \Gamma, H)$ , where  $N$  is the associated nonterminal,  $P$  is the associated production,  $\Gamma$  is an environment that contains bindings for the functions that are mentioned in the rules of  $P$ , and the heap  $H$  contains values for attributes and nodes that form the children of  $P$ .

More precisely, a heap  $H$  is a partial map between descriptors  $d$  and a value  $v$ . A descriptor is either an identifier for an attribute or a child:

$$\begin{array}{ll}
 v ::= v \mid \iota \mid n \mid \ell & \text{-- atomic value } v, \text{ or unit value } \iota, \text{ or node } n, \text{ or reference } \ell \\
 n ::= (N, P, \Gamma, H) & \text{-- node associated to } N \text{ and } P, \text{ with env } \Gamma, \text{ and heap } H \\
 H ::= \emptyset \mid H, d \mapsto_{\delta} v & \text{-- heap of a node (partial map from descriptor to value)} \\
 \Psi ::= \emptyset \mid \Psi, \ell \mapsto v & \text{-- threaded heap (partial map from reference to value)} \\
 \Upsilon ::= \emptyset \mid \Upsilon, \ell & \text{-- set of references } \bar{\ell}
 \end{array}$$

A value of an attribute is either an atomic value  $v$  in the domain of the attribute, or a tree in case of a higher-order attribute.

**Definition** (Normal form). A tree  $n$  is in normal form when the synthesized attributes of the root have been computed, thus when the root node  $n$  has a heap with bindings for all  $\text{syn.lhs}.x$  where  $\text{syn}.x$  is a synthesized attribute declared for the nonterminal of  $n$ .

The small-step relation  $\Gamma_0 \vdash \Psi; v \longrightarrow v'; \Psi'$  (Figure 4.4) describes how to evaluate a tree  $v$  one step further to  $v'$  in a global environment  $\Gamma_0$ . The threaded heap  $\Psi$  contains bindings for references, and is threaded through the evaluation.

To reduce a tree, we start with an initial tree and gradually grow the tree by modifying the heaps. In the initial tree  $v_0 = (N, P, \Gamma, \emptyset)$ ,  $N$  is the nonterminal of the root,  $P$  is the production of the root, and  $\Gamma$  contains bindings for functions as mentioned in the rules of  $P$ . In particular, for each child-rule in  $P$ , there is a binding for a function in  $\Gamma$  that provides the semantics for that child. When the AG is well-formed, the resulting tree is in normal form when the rules are applied exhaustively.

We use the following notation in the rules. The operator  $\cup$  represents the left-biased union of its two operands heaps, and operator  $\uplus$  takes the union of two heaps with disjoint keys. The domain  $\text{dom}(H)$  of a heap  $H$  is the set of the heap's keys. A heap  $H$  applied to a designator  $d$ , written  $H(d)$ , returns the binding for  $d$  in  $H$ . Similarly,  $\Gamma_0 \Gamma_1 f$  returns the function binding for  $f$  in the right-biased union of  $\Gamma_0$  and  $\Gamma_1$ . Juxtaposition of heaps represents the left-biased union.

In Figure 4.4, rule E.DECENT incorporates the idea of nondeterministically selecting a node constructed so far, and reducing it a bit further. Each node has its own heap that explicitly tracks which attributes have been computed, which children have been constructed, and which rules have been applied. A test to check if a rule can be applied then boils down to verifying that their dependencies are met (Figure 4.5).

With Rule E.INH bindings for the inherited attribute of a child can be copied to the heap of that child when the child is ready to receive these bindings. For example, inherited attributes

#### 4 AGs with Commuting Rules

$$\boxed{\Gamma \vdash \Psi_0; v_0 \longrightarrow v_1; \Psi_1}$$

$$\frac{H(\mathbf{child} \ x) = v_0 \quad \Gamma_0 \vdash \Psi_0; v_0 \longrightarrow v_1; \Psi_1}{\Gamma_0 \vdash \Psi_0; (N, P, \Gamma_1, H) \longrightarrow (N, P, \Gamma_1, \{\mathbf{child} \ x \mapsto v_1\} \uplus H); \Psi_1} \text{E.DESCENT}$$

$$\frac{H(\mathit{inh}.x.y) = v \quad H(\mathbf{child} \ x) = (N', P', \Gamma_2, H') \quad N'; P'; H' \sqrt{\mathit{inh}.lhs.y}}{\Gamma_0 \vdash \Psi; (N, P, \Gamma_1, H) \longrightarrow (N, P, \Gamma_1, \{\mathbf{child} \ x \mapsto (N', P', \Gamma_2, \{\mathit{inh}.lhs.y \mapsto v\} \uplus H')\} \uplus H); \Psi} \text{E.INH}$$

$$\frac{H(\mathbf{child} \ x) = (N', P', \Gamma_2, H') \quad H'(\mathit{syn}.lhs.y) = v \quad N; P; H \sqrt{\mathit{syn}.x.y}}{\Gamma_0 \vdash \Psi; (N, P, \Gamma_1, H) \longrightarrow (N, P, \Gamma_1, \{\mathit{syn}.x.y \mapsto v\} \uplus H); \Psi} \text{E.SYN}$$

$$\frac{(\mathbf{child} \ x : N' = f(\overline{z^{\mathcal{D}}})) \in \mathit{rules} \ (N, P) \quad N; P; H \sqrt{\mathbf{child} \ x} \quad \overline{\Psi}; \overline{H} \vdash \overline{z^{\mathcal{D}}} \uparrow v}{\Gamma_0 \vdash \Psi; (N, P, \Gamma_1, H) \longrightarrow (N, P, \Gamma_1, \{\mathbf{child} \ x \mapsto (\Gamma_0 \Gamma_1 f) \overline{v}\} \uplus H); \Psi} \text{E.CHILD}$$

$$\frac{(x : p(\overline{z_1^{\mathcal{D}}}) = f(\overline{z_2^{\mathcal{D}}})) \in \mathit{rules} \ (N, P) \quad N; P; H_0 \sqrt{\mathbf{rule} \ x} \quad \overline{\Psi}_0; H_0 \vdash \overline{z_2^{\mathcal{D}}} \uparrow v_0 \quad \overline{v}_1 = (\Gamma_0 \Gamma_1 p \cdot \Gamma_0 \Gamma_1 f) \overline{v}_0}{\overline{H}_0 \vdash v_1 \downarrow z_1^{\mathcal{D}} \rightsquigarrow H; \overline{\Psi}; \overline{\Upsilon} \quad \overline{\Upsilon}_1 \uplus \dots \uplus \overline{\Upsilon}_n \uplus \mathit{dom} \ \overline{\Psi} = \overline{\ell}}{\Gamma_0 \vdash \Psi_0; (N, P, \Gamma_1, H_0) \longrightarrow (N, P, \Gamma_1, \overline{H} \uplus \{\mathbf{rule} \ x \mapsto \iota\} \uplus H_0); \overline{\Psi} \Psi_0} \text{E.EVAL}$$

$$\frac{\mathit{syn}.y \in \mathit{barriers} \ N \quad N; P; H \sqrt{\mathit{syn}.lhs.y}}{\Gamma_0 \vdash \Psi; (N, P, \Gamma_1, H) \longrightarrow (N, P, \Gamma_1, \{\mathit{syn}.lhs.y \mapsto \iota\} \uplus H); \Psi} \text{E.BAR.LHS}$$

$$\frac{(\mathbf{child} \ x : N' = f(\overline{z^{\mathcal{D}}})) \in \mathit{rules} \ (N, P) \quad \mathit{inh}.y \in \mathit{barriers} \ N' \quad N; P; H \sqrt{\mathit{inh}.x.y}}{\Gamma_0 \vdash \Psi; (N, P, \Gamma_1, H) \longrightarrow (N, P, \Gamma_1, \{\mathit{inh}.x.y \mapsto \iota\} \uplus H); \Psi} \text{E.BAR.INH}$$

**Figure 4.4:** Small-step evaluation rules for a node.

for a child can be computed even before the child is introduced. Rule E.SYN represents the inverse for synthesized attributes. Note the careful use of disjointed unions which ensure that a rule is only applicable once per attribute.

With rules E.CHILD, E.EVAL, E.BAR.INH, and E.BAR.SYN, bindings can be added to the heap. Evaluation of the RHS of the AG's child-rule gives us the initial tree to use for that child in the heap. Evaluation of the AG's evaluation rule results in bindings for one or more attributes. The rule E.EVAL ensures in addition that potentially newly introduced references  $\Upsilon$  do not clash with existing references. The rule E.BAR.INH defines an inherited barrier attribute of a child  $x$ . Rule E.BAR.SYN defines a synthesized barrier attribute of  $lhs$ .

Figure 4.5 shows load, store, and ready rules. The relation  $\Psi; H \vdash z^{\mathcal{D}} \uparrow v$  represents a load of value  $v$  using designator  $z$  in heap  $H$ . The load rule L.VAL requires the presence of a designator in the heap. With the rule the value of the designator can be extracted from the head. The load rule L.REF represents an indirect load from the threaded heap  $\Psi$ .

The relation  $H_0 \vdash v \downarrow z \rightsquigarrow H; \Psi; \Upsilon$  represents a store of value  $v$ . The heap  $H$  and treaded heap  $\Psi$  contain the new bindings that result from storing  $v$ . The initial heap  $H_0$  is stores

$$\begin{array}{c}
\boxed{\Psi; H \vdash z^\circ \uparrow v} \\
\frac{H(z) = v}{\Psi; H \vdash z^\bullet \uparrow v} \text{L.VAL} \qquad \frac{H(z) = \ell \quad \Psi(\ell) = v}{\Psi; H \vdash z^\circ \uparrow v} \text{L.REF} \\
\boxed{H_0 \vdash v \downarrow z^\triangleleft \rightsquigarrow H; \Psi; \Upsilon} \\
H_0 \vdash v \downarrow z^x \rightsquigarrow \{z \mapsto \ell\}; \{\ell \mapsto v\}; \{\ell\} \text{S.NEW} \qquad H_0 \vdash v \downarrow z^\bullet \rightsquigarrow \{z \mapsto v\}; \emptyset; \emptyset \text{S.SAVE} \\
\frac{H(z) = \ell}{H_0 \vdash v \downarrow z^\circ \rightsquigarrow \emptyset; \{\ell \mapsto v\}; \emptyset} \text{S.UPDATE} \\
\boxed{N; P; H \sqrt{d}} \\
\frac{N; P; H \sqrt{\text{dep } d} \quad \{d' \mid (\text{order } d' \prec d) \in \text{rules}(N, P)\} \subseteq \text{dom } H}{N; P; H \sqrt{d}} \text{D.ORDER} \\
\frac{(x: p \overline{z_1^\triangleleft} = f \overline{z_2^\triangleright}) \in \text{rules}(N, P)}{N; P; H \sqrt{z_1}} \text{D.R} \qquad \frac{}{N; P; H \sqrt{\text{dep } \text{child } x}} \text{D.C} \qquad \frac{}{N; P; H \sqrt{\text{dep } z}} \text{D.A} \\
N; P; H \sqrt{\text{dep } \text{rule } x} \qquad N; P; H \sqrt{\text{dep } \text{child } x} \qquad N; P; H \sqrt{\text{dep } z}
\end{array}$$

**Figure 4.5:** Load, store and designator dependency rules.

the reference when  $z^\triangleleft$  is an indirection. The set of references  $\Upsilon$  contains the newly added references. The store rule `S.NEW` represents the creation and assignment of a new reference  $\ell$ , and `S.UPDATE` an update via such a reference. With rule `S.SAVE` a binding for  $z$  is added without an indirection.

The relation  $N; P; H \sqrt{d}$  describes when  $d$  is ready to be defined while taking dependency-rules into account. These rules do not test for the existence of values in the heap, because the load and store rules already cover for this. The rule `D.ORDER` handles dependency-rules generically for the designators. The rule `D.R` states that an AG rule is ready when its LHS is ready.

**Remarks.** If we can determine for a nonterminal that no subtree applies rule `S.NEW` or `S.UPDATE`, only topdown behavior for  $\Psi$  is needed. If in addition `L.REF` cannot occur, the two  $\Psi$  parameters can be omitted from the relation.

The rule `E.DESCENT` facilitates a walk down the tree until the node to reduce is reached. An actual implementation performs multiple evaluation steps (if possible) on such a node in order to reduce traversal overhead. In fact, it is possible to statically analyze the AG and obtain a static scheduling of the rules.

$$\begin{aligned}
 \llbracket \mathbf{sem} P : N \bar{r} \rrbracket_{\text{vert}} &= \{ \mathbf{child} \text{ lhs} \} \cup \llbracket \bar{a}_N \rrbracket_{\text{vert}}(\text{lhs}) \cup \llbracket \bar{r} \rrbracket_{\text{vert}} \\
 \llbracket \mathbf{inh} y t \rrbracket_{\text{vert}(x)} &= \{ \text{inh}.x.y \} \\
 \llbracket \mathbf{syn} y t \rrbracket_{\text{vert}(x)} &= \{ \text{syn}.x.y \} \\
 \llbracket \mathbf{child} x : N = f \bar{z}^{\triangleright} \rrbracket_{\text{vert}} &= \{ \mathbf{child} x \} \cup \llbracket \bar{a}_N \rrbracket_{\text{vert}(x)} \\
 \llbracket x : p \bar{z}_1^{\triangleleft} = f \bar{z}_2^{\triangleright} \rrbracket_{\text{vert}} &= \{ \mathbf{rule} x \} \\
 \llbracket \mathbf{order} d_1 < d_2 \rrbracket_{\text{vert}} &= \emptyset \\
 \\ 
 \llbracket \mathbf{sem} P : N \bar{r} \rrbracket_{\text{edge}} &= \llbracket \bar{r} \rrbracket_{\text{edge}} \\
 \llbracket \mathbf{child} x : N = f \bar{z}^{\triangleright} \rrbracket_{\text{edge}} &= \overline{z \leftarrow \mathbf{child} x} \cup \{ \mathbf{child} x \leftarrow \mathbf{syn}.x.y \mid \mathbf{syn}.y \in a_N \} \\
 \llbracket x : p \bar{z}_1^{\triangleleft} = f \bar{z}_2^{\triangleright} \rrbracket_{\text{edge}} &= \overline{z_2 \leftarrow \mathbf{rule} x} \cup \llbracket z_1^{\triangleleft} \rrbracket_{\text{pre}(x)} \\
 \llbracket \mathbf{order} d_1 < d_2 \rrbracket_{\text{edge}} &= \{ d_1 \leftarrow d_2 \} \\
 \llbracket z^{\bullet} \rrbracket_{\text{pre}(x)} &= \{ \mathbf{rule} x \leftarrow z \} \\
 \llbracket z^{\times} \rrbracket_{\text{pre}(x)} &= \{ \mathbf{rule} x \leftarrow z \} \\
 \llbracket z^{\circ} \rrbracket_{\text{pre}(x)} &= \{ z \leftarrow \mathbf{rule} x \}
 \end{aligned}$$

**Figure 4.6:** Vertices and edges of the initial production dependency graph.

## 4.4 Static Dependency Graphs

In order to obtain a static scheduling of the rules, we first construct dependency graphs in the style of Knuth-1 [Knuth, 1968]. If these graphs are cycle free, a static scheduling of the rules is possible.

Unlike in Section 1.3.2, we mean an augmented PDG when we talk about a PDG. When we talk about the initial PDG, we mean the non-augmented PDG as in Section 1.3.2. Thus, we will be dealing with dependency graphs per production (augmented production dependency graph  $pdg(N, P)$ ) and dependency graphs per nonterminal (a nonterminal dependency graph  $ndg(N)$ ).

We first define the initial graphs, then specify the actual graphs as the fixpoints of the functions:

$$\begin{aligned}
 pdg N P &= pdg_{\text{init}} N P \cup \{ \text{instantiate } x (ndg N_x) \mid x \in \text{children } P \} \\
 ndg N &= ndg_{\text{init}} N \cup \{ \text{abstract } (pdg N P) \mid P \in \text{prods } N \}
 \end{aligned}$$

The function *instantiate* translates the edges between declarations of  $N_c$  in the NDG as edges between the attributes of child  $c$  in the PDG. Similarly, when there is a path between two attributes of *lhs* in the PDG, then *abstract* translates these as edges between the same attributes in  $N$ 's NDG.

**Initial production dependency graph.** Per production  $P$  of nonterminal  $N$ , we construct a production dependency graph  $pdg_{\text{init}} N P$ . Thus, each sem-block is translated to a PDG.

Figure 4.6 shows that the vertices of  $pdg_{\text{init}} N P$  consists of designators  $\llbracket \mathbf{sem} P : N \bar{r} \rrbracket_{\text{vert}}$ . In this notation,  $\bar{a}_N$  is the set of attribute declarations  $a$  of nonterminal  $N$ . The directed edges  $\llbracket \mathbf{sem} P : N \bar{r} \rrbracket_{\text{edge}}$  of  $pdg(N, P)$  relate designators. An edge from  $d_2$  to  $d_1$  denotes that  $d_1$  must be defined before  $d_2$ . Order rules are thus a means to arbitrarily add edges to the PDG.

The initial PDG does not contain the dependencies imposed by the semantics of each child (e.g. the actual tree). We obtain the actual production dependency graph by augmenting it with the edges taken from the nonterminal dependency graphs of the children which are an approximation of the dependencies of such trees.

**Initial nonterminal dependency graph.** The vertices of the nondeterminal dependency graph  $ndg(N)$  are the declarations  $a$  of  $N$ :

$$\begin{aligned} \llbracket \mathbf{attr} N \bar{a} \rrbracket_{\text{vert}} &= \llbracket \bar{a} \rrbracket_{\text{vert}} \\ \llbracket \mathbf{inh} x t \rrbracket_{\text{vert}} &= \{ inh.x \} \\ \llbracket \mathbf{syn} x t \rrbracket_{\text{vert}} &= \{ syn.x \} \end{aligned}$$

The initial NDG does not have any edges. These edges are inferred from the PDGs, as we see below.

**Graph construction.** The actual PDG is the least solution to the above equations with the following definitions for *abstract* and *instantiate*. The relation  $d_1 \leftarrow^+ d_2$  represents a path from  $d_2$  to  $d_1$ , with  $d_1 \neq d_2$ :

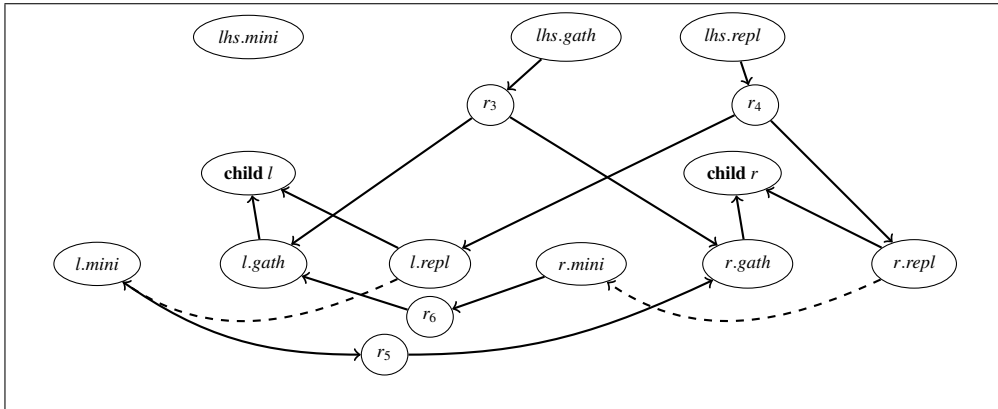
$$\begin{aligned} \mathit{abstract} g &= \{ k_1.x \leftarrow k_2.y \mid k_1.lhs.x \leftarrow^+ k_2.lhs.y \in g \} \\ \mathit{instantiate} x g &= \{ k_1.x.y \leftarrow k_2.x.z \mid k_1.y \leftarrow^+ k_2.z \in g \} \end{aligned}$$

Conventionally, a synthesized attribute of a child only depends on an inherited attribute of a child. However, due to dependency-rules, any attribute of a child can potentially depend on any other attribute of the child, hence our definition of *abstract* and *instantiate* take these also into account.

The construction of the graphs is a relatively straightforward fixpoint computation. As intermediate data structure, a transitively closed PDG allows for efficient tests for paths between vertices.

**Example.** In the Leaf-production, there is dependency of *syn.lhs.repl* on *inh.lhs.mini* via rule  $r_2$ . This thus induces a dependency of *syn.repl* on *inh.mini* in the NDG. Figure 4.7 shows the PDG of the production *Bin*. The solid edges are initial edges, and the dashed edges are instantiated from the NDG. The Bin-production by itself does not induce any edges in the NDG. We do not show the PDG of the Leaf-production here; it is shown in Figure 4.15(a).

**Properties.** A NDG can only be cyclic if at least one of the PDGs is cyclic. The reason is that the edges of the NDG are projected into the PDGs, thus a PDG must have a subgraph with these cyclic edges.



**Figure 4.7:** Example of the (augmented) PDG of the Bin-production.

When the PDGs are acyclic, then an evaluation to normal form is possible for any well-formed tree. In other words, the graph construction is sound. This property has a relatively straightforward structural induction proof on the shape of the tree. The NDGs symbolize the induction hypothesis. When an evaluation algorithm respects the dependencies of the PDGs, and the constraints of each rule are preserved, the evaluation algorithm is sound. Moreover, the evaluation algorithm is complete if it finds a scheduling when the PDGs are acyclic. We show such an algorithm in Section 4.5.

Rules of a production may make an inherited attribute of a child dependent on a synthesized attribute of a child. These dependencies are not part of the NDG. When we add these edges also to the NDG, each occurrence of a nonterminal has the same dependencies of its inherited attributes on its synthesized attributes. When the PDGs are cycle free without these edges, they are typically also cycle free with these additional edges. In the next section, it becomes clear that adding these edges may reduce the number of visit interfaces that a nonterminal symbol needs to support, thus can be beneficial when the code size is an issue.

## 4.5 Visits Graphs

An interface for a nonterminal is a partitioning of the attributes into a finite sequence of visits. Given such a sequence, we can produce an execution plan and generate code, as we showed in Chapter 3. We derive such interfaces from acyclic PDGs.

Given such a sequence, it induces a dependency of all attributes of a later visit on attributes of an earlier visit. When represented as scheduling-induced edges in the PDG, it may make the PDG cyclic. It is typically possible to define a single visit sequence per nonterminal that does not make the PDG cyclic. However, it is generally hard to automatically find such a sequence [Kastens, 1980] when the sequence is not manually specified as we presented in Chapter 3. Instead, if we allow multiple interfaces for a nonterminal, we can define visit sequences so that they do not lead to additional dependencies. This is the approach taken by Kennedy and Warren [1976].

$$\begin{aligned}
\llbracket \mathbf{visit} \ i \ \overline{k.y} \rrbracket_{\text{vert}(x)} &= \{ \mathbf{visit} \ x \ i \} \\
\llbracket \mathbf{visit} \ i \ \overline{k.y} \rrbracket_{\text{edge}(x)} &= \llbracket i \rrbracket_{\text{edge}(x, \text{pre } i)} \cup \llbracket \overline{k.y} \rrbracket_{\text{edge}(x, i)} \\
\llbracket i \rrbracket_{\text{edge}(x, \varepsilon)} &= \{ \mathbf{child} \ x \} \\
\llbracket i \rrbracket_{\text{edge}(x, j)} &= \{ \mathbf{visit} \ x \ j \leftarrow \mathbf{visit} \ x \ i \} \\
\llbracket inh.y \rrbracket_{\text{edge}(x, i)} &= \{ inh.x.y \leftarrow \mathbf{visit} \ x \ i \} \\
\llbracket syn.y \rrbracket_{\text{edge}(x, i)} &= \{ \mathbf{visit} \ x \ i \leftarrow syn.x.y \}
\end{aligned}$$

**Figure 4.8:** The embedding of a context.

In our experience, Kastens' approach requires severe manual intervention for large AGs, thus we take the approach by Kennedy and Warren [1976]. However, it is not immediately obvious how to deal with multiple interfaces in a strongly typed language, because the type of the semantic function depends on the interface. In the next section, we solve this problem using type indices in combination with GADTs. In this section, we show how to determine these interfaces.

**Contexts.** A context  $\mathcal{C}$  of a nonterminal  $N$ , or nonterminal symbol with  $N$ , or a production of  $N$  is a visit sequence (many  $v$ ) that is consistent with  $ndg \ N$ :

$$\begin{aligned}
\mathcal{C} &::= \overline{v} && \text{-- a context consists of a sequence of visits } \overline{v} \\
v &::= \mathbf{visit} \ i \ \overline{k.x} && \text{-- a visit consists of a set of attributes } \overline{k.v} \text{ and globally unique } i \\
i &&& \text{-- identifier}
\end{aligned}$$

A context contains a subset of the attributes declared for a nonterminal. A nonterminal may have a context with less attributes when not all attributes of a nonterminal symbol with this nonterminal are needed.

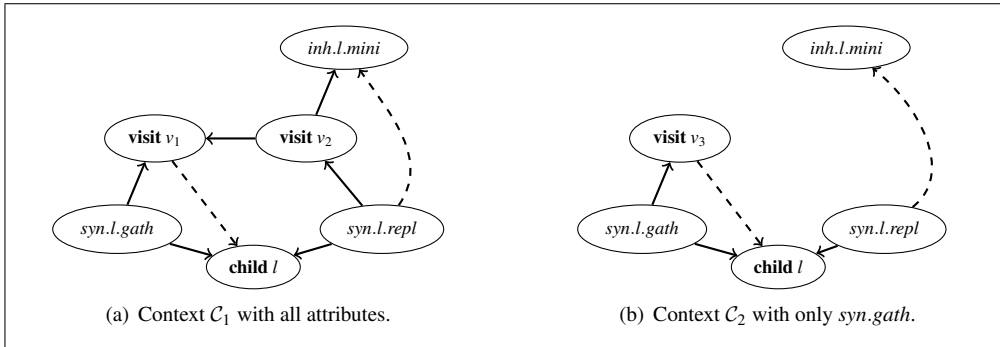
A visit sequence is consistent with  $ndg \ N$  when the following conditions are met.

- For each attribute  $inh.x$  and  $syn.y$  in the sequence, with  $inh.x \leftarrow^+ syn.y$  in  $ndg \ N$ , either  $syn.y$  is not part of the visit sequence, or  $syn.y$  is in the same or later visit as  $inh.x$ .
- For each attribute  $syn.x$  and  $inh.y$  in the sequence, with  $syn.x \leftarrow^+ inh.y$  in  $ndg \ N$ , either  $inh.y$  is not part of the visit sequence, or  $inh.y$  is in a later visit as  $syn.x$ .

The number of consistent visit sequences is finite, but potentially large, as it is in the worst case exponential in the number of attributes.

**Embedding of a context.** We make the context of a child visible in the PDG. For that purpose, we introduce a vertex form that represents a visit:

$$\begin{aligned}
d &::= \dots \\
| \ \mathbf{visit} \ x \ i & \text{ -- a visit to } x \text{ identified by } i
\end{aligned}$$



**Figure 4.9:** Examples of visit-additions to the PDG.

For a child  $x$  in context  $C_x$ , we add additional vertices and edges to the PDG, which we call the *embedding* of  $C_x$  of child  $x$  in the PDG. Informally, to embed a context, we add a visit-node and add edges between this node and the associated inherited and synthesized attributes. Additionally, a visit node is dependent on the node of the preceding visit, if any, and the child-node otherwise. Formally, let  $pre\ i$  either be the preceding visit  $j$  of  $i$ , or  $\varepsilon$  when  $i$  is the first visit. We add vertices  $\llbracket C \rrbracket_{vert(x)}$  and edges  $\llbracket C \rrbracket_{edge(x)}$  as described in Figure 4.8.

**Definition** (Visit PDG). For some given contexts of children, a *visit PDG* (VPDG) is the embedding of these contexts in the PDG.

The following are exemplary contexts of child  $l$  of production *Bin*. The context  $C_1$  describes the computation of all attributes in two phases. The context  $C_2$  only the computation of *syn.gath*:

$$\begin{aligned} C_1 &= [\mathbf{visit}\ v_1\ \{syn.gath\}, \mathbf{visit}\ v_2\ \{inh.mini, syn.repl\}] \\ C_2 &= [\mathbf{visit}\ v_3\ \{syn.gath\}] \\ C_3 &= [\mathbf{visit}\ v_4\ \{syn.gath, inh.mini, syn.repl\}] \end{aligned}$$

Figure 4.9 shows the relevant subgraphs after embedding the respective contexts  $C_1$  and  $C_2$ . The dashed edges are already existing edges. The embedding of  $C_3$  would lead to a cycle in the PDG.

A visit-node in a VPDG serves as bookkeeping that a visit to the child is needed to compute the attributes it is associated with. It represents dependencies induced from the AG-wide scheduling of attribute computations, and we will ensure later that each visit node has an associated algorithm that computes the attributes.

**Abstraction of visit-nodes.** We see later as well that the bookkeeping with visits is too verbose, so we establish a more compact representation. For that purpose, we change the syntax of child-vertices. These additionally define in what state  $\bar{a}$  the child is, which can be used in the graph to *abstract* from a sequence of visits:

$$\begin{aligned} d ::= & \dots \\ & | \mathbf{child}\ x\ \bar{a} \quad \text{-- a child } x \text{ in state } \bar{a} \text{ (initially } \emptyset) \end{aligned}$$



$$\begin{array}{l}
\text{abstract } g \\
| \langle \text{invoke } x i \rangle \in g \wedge \langle \text{child } x \bar{a} \leftarrow \text{invoke } x i \rangle \in g = \\
\text{let } v = \langle \text{child } x (\bar{a} \uplus \bar{a}_i) \rangle \\
\text{es} = \{ v \leftarrow \text{syn}.x.y \mid \text{syn}.x.y \in g \} \cup \\
\quad \{ v \leftarrow \text{invoke } x j \mid \langle \text{invoke } x j \rangle \in g, \langle \text{invoke } x i \leftarrow \text{invoke } x j \rangle \in g \} \\
\text{in } \text{abstract} ((g \cup \{v\} \cup \text{es}) - \{ \text{invoke } x i, \text{child } x \bar{a} \}) \\
| \text{otherwise} = g
\end{array}$$

Figure 4.10: Abstraction on a VPDG.

Let  $g$  be an acyclic VPDG. Figure 4.10 shows how to abstract from the visit-vertices. We first remove all visit-vertices. When a visit-vertex  $i$  is removed, its associated attributes  $\bar{a}_i$  are added to the child-vertex. There are no visit-vertices left in  $g$  after *abstract* has been applied.

With abstraction applied to a VPDG we obtain scheduled PDGs:

**Definition** (Scheduled PDG). Given a production  $P$  in context  $\mathcal{C}$ , and context  $\bar{\mathcal{C}}$  for its children, its VPDG  $g$  is a *scheduled PDG* (SPDG) when:

- The VPDG  $g$  is acyclic.
- All reachable synthesized attributes of a child depend on a child node that includes the synthesized attribute in its state. Thus, for each  $\text{syn}.y \in \mathcal{C}$ , for each path in the VPDG from vertex  $\text{syn}.lhs.y$  to a vertex  $\text{syn}.x.z$ , there exists a vertex **child**  $x \bar{a}$  with  $\text{syn}.z \in \bar{a}$ .

These two conditions allow us to gradually determine contexts, as we show in the remainder of this section. We show in the next section how to convert a SPDG to an execution plan for a production.

**Representation.** We determine a set of contexts for each nonterminal, such that there exists a SPDG for each context and each production. Preferably, the set of contexts is small, because each context requires a different semantic function. Also, to save traversal overhead, preferably each context contains visits with many attributes. To determine this set, we use a more sophisticated representation, the visits graph, which we define below.

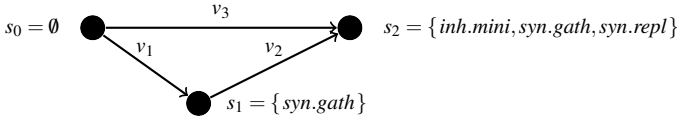
Note again that a context is a sequence of visits that is associated with a nonterminal  $N$ , where each visit describes a transition of the configuration of a node associated with  $N$ . Given a set of contexts, sequences of visits may have common intermediate configurations. Thus, we can represent a set of contexts as a graph, which we call the visits graph:

**Definition** (Visits graph). Given a set of contexts  $\bar{\mathcal{C}}$ , a *visits-graph* is DAG where the vertices are the union of  $\llbracket \mathcal{C} \rrbracket_{\text{vert}}$  of each  $\mathcal{C} \in \bar{\mathcal{C}}$ , and the edges are the union of  $\llbracket \mathcal{C} \rrbracket_{\text{edge}}$  of each  $\mathcal{C} \in \bar{\mathcal{C}}$ , as described by Figure 4.11. The vertices represent configurations, and the edges represent visits.

#### 4 AGs with Commuting Rules

$$\begin{aligned}
\llbracket \mathcal{C} \rrbracket_{\text{vert}} &= \llbracket \mathcal{C} \rrbracket_{\text{vert}}(\emptyset) \\
\llbracket \emptyset \rrbracket_{\text{vert}}(s) &= \{s\} \\
\llbracket \text{visit } i \bar{a} : vS \rrbracket_{\text{vert}}(s) &= \{s\} \cup \llbracket vS \rrbracket_{\text{vert}}(s \cup \bar{a}) \\
\llbracket \mathcal{C} \rrbracket_{\text{edge}} &= \llbracket \mathcal{C} \rrbracket_{\text{edge}}(\emptyset) \\
\llbracket \emptyset \rrbracket_{\text{edge}}(s) &= \emptyset \\
\llbracket \text{visit } i \bar{a} : vS \rrbracket_{\text{edge}}(s) &= \{s \xrightarrow{i} (s \cup \bar{a})\} \cup \llbracket vS \rrbracket_{\text{edge}}(s \cup \bar{a})
\end{aligned}$$

**Figure 4.11:** Vertices and edges of the visits-graph.



**Figure 4.12:** Example of a visits-graph which represents contexts  $\mathcal{C}_1$ ,  $\mathcal{C}_2$  and  $\mathcal{C}_3$ .

Figure 4.12 shows the visits-graph that represents the contexts  $\mathcal{C}_1$ ,  $\mathcal{C}_2$  and  $\mathcal{C}_3$ . Note that the context  $\mathcal{C}_2$  is fully implied by the other contexts.

For a generated algorithm, a path in the visits-graph represents how a node is visited by its parent. However, a parent may stop invoking visits after any visit. From a code generation perspective, it may be beneficial to know at which vertex a parent stops. We can model this situation in the graph, but we refrain from this complication here.

When two paths in the visits graph converge, this means that two different visit sequences ended up in the same configuration. Consequently, the children are in the same configuration, but were potentially subjected to different visit sequences. The VPDGs may thus differ in the visit-vertices, therefore we need *abstract* (as defined earlier) to eliminate this difference.

We keep a more complex administration for the visits graph, with vertices of the form  $s$  and edges of the form  $v$ , as described by Figure 4.13. We explain some aspects of this notation below. Figure 4.14 shows this administration for the example.

A vertex of the visits graph represents a context. Per production, a vertex stores the VPDG in this context, and the states of the children. An edge represents a visit, which is a transition from one configuration to one of its next configurations. It stores the attributes  $\bar{a}$ , which are the inherited attributes provided by the parent for the visit, and the synthesized attributes that need to be computed for the parent. Per production, an edge stores an ordered sequence of invocations to children that are required for the visit. These visits are grouped in what we call *simultaneous visits*. Visits of such a group visit children independently. Note that these visits may be invoked in any order, and may also be invoked concurrently.

Figure 4.15(a) displays the graphs  $g_1$ ,  $g_2$ ,  $g_3$  and  $g_4$  that are mentioned in Figure 4.14. The operation *abstract* is not yet applied in the graphs. The dotted edges represent the edges that are inserted due to the embedding of visits. An important property of such a graph is that the

$s ::= \mathbf{conf} N \bar{a} \bar{p}$	-- configuration of a node with nonterminal $N$ (key $\bar{a}$ )
$p ::= \mathbf{prod} P \bar{k} g$	-- production in a context of nonterminal $N$
$k ::= \mathbf{child} x : N \bar{a}$	-- configuration of child $x$ (key $\bar{a}$ )
$e ::= s ; v ; s'$	-- edge between $s$ and $s'$
$v ::= \mathbf{visit} i : N \bar{a} \bar{c}$	-- visit $i$ with production info $\bar{c}$ (key $i$ , non-empty $\bar{a}$ )
$c ::= \mathbf{prod} P \bar{r}$	-- sequence of invocation sets $\bar{r}$ (possibly empty)
$r ::= \mathbf{sim} \bar{m}$	-- simultaneous invocations $\bar{m}$ (non-empty, unordered)
$m ::= \mathbf{invoke} x i$	-- invocation of visit $i$ to $x$ (key $i$ )
$g$	-- VPDG of a production in a given context
$N, P$	-- identifier (nonterminal), identifier (production)

**Figure 4.13:** Notation for visits-graphs.

simultaneous visits contain precisely those synthesized attributes that can be visited because the inherited attributes are available (described below with the relation *avail*), and also need to be visited in order to produce values for the synthesized attributes.

The example is rather symmetric. The children of the production *Bin* are treated in the same way. This is in general not the case and will be handled correctly. For example, children may be of different nonterminals, or may be visited in a different order. This situation arises when we use  $\langle \mathbf{invoke} l 3 \rangle$  for the left child of the bin-production, or in the example of Section 2.2.

**Invariants.** The representation of visits graph leads to a recursive set of constraints (presented below) for which we can incrementally construct a visits graph. For a given attribute grammar, there may be many possible visits graphs. The way we represent the graph gives rise to visit sequences that are relatively independent and only compute what is necessary, but which are not guaranteed to be as large as possible. Different visits graphs may exhibit small differences in performance, although in principle any visits graph suffices. We impose a number of constraints on the visits graph for which it is possible to find a solution when the PDGs are acyclic, and for which it is possible to find a *unique* solution.

In the formalization below, we denote a relation as partial function without a right-hand side, or as (partial) boolean functions. Moreover, when used in a boolean expression, we consider such relations a total function from the arguments to a Boolean result value, which is *True* if and only if the arguments form an element of the relation. When the expression of a guard has the value  $\perp$ , then the guard itself has the value *False*. Also, for unbound variables in pattern expressions we assume existential quantification, unless indicated explicitly otherwise.

**Definition (Available).** A vertex  $d$  is *available* in a VPDG  $g$  when vertex  $d$  can be scheduled, and respects the order imposed by  $g$ . Formally, the relation *avail*  $\bar{a} g d$  as defined in Figure 4.16 (explained below) states whether a vertex  $d$  in a VPDG  $g$  is available when the node is in configuration  $\bar{a}$ .

```

s0 = conf Tree 0 {p1,p2}
s1 = conf Tree {syn.gath} {p3,p4}
s2 = conf Tree {inh.mini,syn.gath,syn.repl} {p5,p6}

p1 = prod Leaf 0 g1
p3 = prod Leaf 0 g1
p5 = prod Leaf 0 g1
p2 = prod Bin {child l:Tree 0,child r:Tree 0} (abstract g2)
p4 = prod Bin {child l:Tree {syn.gath},child r:Tree {syn.gath}} (abstract g3)
p6 = prod Bin {child l:Tree {inh.mini,syn.gath,syn.repl}
, child r:Tree {inh.mini,syn.gath,syn.repl}} (abstract g4)

v1 = visit 1:Tree {syn.gath} {c1,c2}
v2 = visit 2:Tree {inh.mini,syn.repl} {c3,c4}
v3 = visit 3:Tree {inh.mini,syn.gath,syn.repl} {c5,c6}

c1 = prod Leaf 0
c3 = prod Leaf 0
c5 = prod Leaf 0
c2 = prod Bin {sim {invoke l 1,invoke r 1}}
c4 = prod Bin {sim {invoke l 2,invoke r 2}}
c6 = prod Bin {sim {invoke l 1,invoke r 1},sim {invoke l 2,invoke r 2}}

```

**Figure 4.14:** The contents of nodes and vertices of the visits graph of the example.

The relation *avail* states that a vertex  $d$  is available when all its dependencies are available, with the exception that only the inherited attributes *inh.lhs.y* are available if they are part of the configuration  $\bar{a}$ , and that synthesized attributes of visits are only available if there was a visit that computed them. Thus, *avail* gives us a notion of which vertices are scheduled in a VPDG in a given configuration.

**Definition (Well-formed).** A visits-graph  $g$  is *well-formed* when it satisfies the invariants below. Additionally, the visits-graph is acyclic, and so are the VPDGs that are stored in the configurations. Also, in each edge of  $g$  or each vertex of  $g$  with some nonterminal  $N$ , there is exactly an entry  $\langle prop P \dots \rangle$  for each production  $P$  of  $N$ . We omit some straightforward structural invariants, such as that each visit-edge is annotated with the same nonterminal  $N$  as the nonterminal  $N$  of the two configurations it connects.

The main invariant is that all visits that are correctly represented in the graph, which is captured by the property in Figure 4.17. A visit-edge of nonterminal  $N$  requires a transformation of the VPDG for each production of  $N$ . The visit must precisely mention the new attributes  $\bar{a}$  that are added to  $a_0$  to form  $a_1$ . The sequence of simultaneous invocations  $\bar{r}$  describes these transitions for the children. When applied to the VPDG  $g_0$  of the old configuration,  $embed \bar{a}_1 g_0 \bar{r}$  gives the unabridged VPDG  $g_1$  of the new configuration. Moreover,  $g_1$  must be complete, which means that all the synthesized attributes *syn.lhs.y* in the new configura-

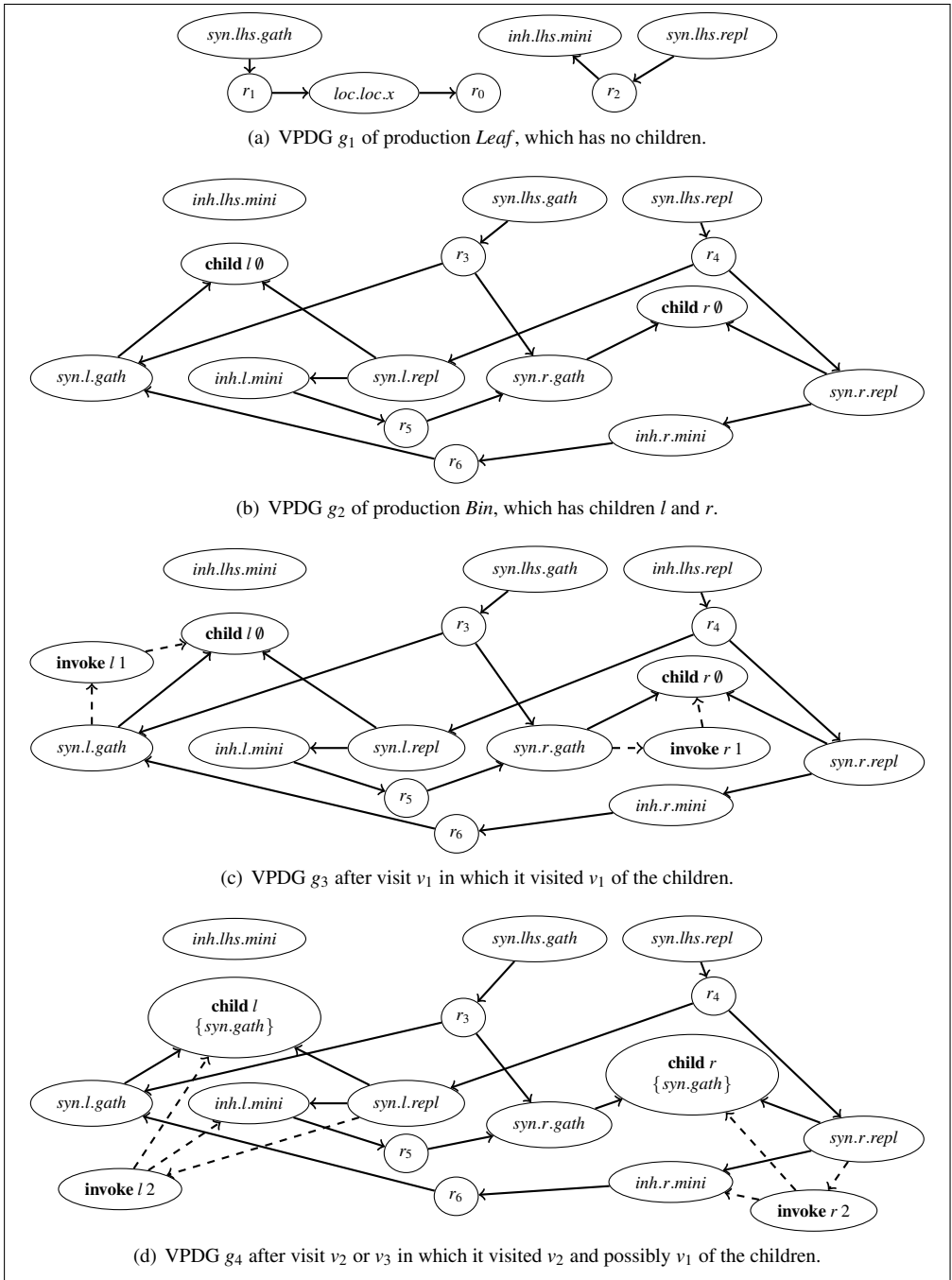


Figure 4.15: VPDGs of the visits graph example.

#### 4 AGs with Commuting Rules

$$\begin{array}{lcl}
\text{avail } \bar{a} \ g \ d & = & \text{all } (\text{avail } \bar{a} \ g) (\text{deps } g \ d) \wedge \text{except } \bar{a} \ g \ d \\
\text{except } \bar{a} \ g \ \langle \text{syn}.x.y \ \rangle & | & \langle \text{visit } x \ i \ \rangle \in \text{deps } g \ \langle \text{syn}.x.y \ \rangle \\
& & | \langle \text{child } x \ \bar{a}' \ \rangle \in \text{deps } g \ \langle \text{syn}.x.y \ \rangle \wedge \langle \text{syn}.x.y \ \rangle \in \bar{a}' \\
\text{except } \bar{a} \ g \ \langle \text{inh}.x.y \ \rangle & | & \text{True} \\
\text{except } \bar{a} \ g \ \langle \text{syn}.lhs.y \ \rangle & | & \text{True} \\
\text{except } \bar{a} \ g \ \langle \text{inh}.lhs.y \ \rangle & | & \text{inh}.y \in \bar{a} \\
\text{except } \bar{a} \ g \ \langle \text{rule } x \ \rangle & | & \text{True} \\
\text{except } \bar{a} \ g \ \langle \text{child } x \ \rangle & | & \text{True} \\
\text{except } \bar{a} \ g \ \langle \text{visit } x \ i \ \rangle & | & \text{True}
\end{array}$$

**Figure 4.16:** The definition of *avail*.

$$\begin{array}{l}
\langle \text{conf } N \ \bar{a}_0 \ \bar{p}_0; \text{visit } i : N \ \bar{a} \ \bar{c}; \text{conf } N \ \bar{a}_1 \ \bar{p}_1 \rangle \in \text{edges } VG_N = \\
\text{all}_3 (\text{transition } \bar{a}_1 \ \bar{a} \ \bar{p}_0 \ \bar{c} \ \bar{p}_1 \wedge \bar{a}_1 \equiv \bar{a}_0 \uplus \bar{a} \\
\text{transition } \bar{a}_1 \ \bar{a} \ (\text{prod } P \ \bar{k}_0 \ g_0) (\text{prod } P \ \bar{r}) (\text{prod } P \ \bar{k}_1 \ g_1) = \\
\text{abstract } (\text{embed } \bar{a}_1 \ g_0 \ \bar{r}) \equiv g_1 \wedge \text{complete } \bar{a}_1 \ \bar{a} \ g_1 \wedge \text{all}_2 (\text{trchild } \bar{r}) \ \bar{k}_0 \ \bar{k}_1 \\
\text{trchild } \bar{r} \ \langle \text{child } x : N \ \bar{a} \ \rangle \langle \text{child } x : N \ \bar{a}' \ \rangle = \bar{a}' \equiv \text{foldl } (\text{trsim } x) \ \bar{a} \ \bar{r} \\
\text{trsim } x \ \bar{a} \ \langle \text{sim } \bar{m} \ \rangle \\
| x \notin \text{children } \bar{m} \quad \quad \quad = \emptyset \\
| \langle \text{invoke } x \ i \ \rangle \in_1 \bar{m} \wedge \langle s; \text{visit } i : N \ \bar{a}' \ \bar{c}; s' \ \rangle \in \text{edges } VG_N = \bar{a} \uplus \bar{a}' \\
\text{children } \bar{m} \quad \quad \quad = \{x \mid \langle \text{invoke } x \ i \ \rangle \in \bar{m}\} \\
\text{complete } \bar{a}_1 \ \bar{a} \ g_1 = \text{all } (\text{avail } \bar{a}_1 \ g_1) \{k.lhs.x \mid k.x \in \bar{a}\}
\end{array}$$

**Figure 4.17:** Main invariant: are all visits represented.

tion must be available in  $g_1$ . Finally, the invocations  $\bar{r}$  also describe the state transitions of the children. The relation  $\text{trchild } \bar{r} \ k_0 \ k_1$  relates the old configuration  $k_0$  of a child to the new configuration  $k_1$ .

The relation  $\text{req } \bar{a} \ g \ d$  in Figure 4.18 specifies if  $d$  is required for the synthesized attributes of  $\bar{a}$  to be *available* in  $g$ . Note that the relation  $d_1 \leftarrow^* d_2$  represents a possibly empty path between  $d_1$  and  $d_2$ .

The function *embed* in Figure 4.18 applies the invocations of visits  $\bar{r}$  to the VPDG  $g$ . A visit may only be invoked if its synthesized attributes are required, since our strategy schedules only the attributes that are needed. Moreover, the inherited attributes must be available.

The simultaneous invokes represent invocations that can be applied in any order. Since we test which inherited attributes are available according to  $g_0$ , the order in which these visits are applied does not affect the PDGs. This is because the synthesized attributes of the child are not available in  $g_0$ , and can thus not be used to make more inherited attributes of another

$$\begin{aligned} req \bar{a} g d &= d \in reqs \bar{a} g \\ reqs \bar{a} g &= \{d \mid inh.x \in \bar{a}, syn.y \in \bar{a}, (inh.lhs.x \leftarrow^* d \leftarrow^* syn.lhs.y) \in g\} \end{aligned}$$

$$\begin{aligned} embed \bar{a} g \langle \bar{r} \rangle &= foldl (embed \bar{a}) g \bar{r} \\ embed \bar{a} g \langle \mathbf{sim} \bar{m} \rangle &= foldl (embed \bar{a} g) g \bar{r} \\ embed \bar{a} g_0 g \langle \mathbf{invoke} x i \rangle \\ &\quad | all (avail g_0 \bar{a}) inhs \wedge all (req g_0 \bar{a}) syns \\ &= \llbracket \mathbf{visit} i \bar{a}' \rrbracket_{vert(x)} \uplus \llbracket \mathbf{visit} i \bar{a}' \rrbracket_{edge(x)} \uplus g \end{aligned}$$

where

$$\begin{aligned} \langle s; \mathbf{visit} i : N \bar{a}' \bar{c}; s' \rangle &\in edges VG_N \\ inhs &= \{inh.x.y \mid inh.y \in \bar{a}'\} \\ syns &= \{syn.x.y \mid syn.y \in \bar{a}'\} \end{aligned}$$

**Figure 4.18:** The relations *req* and *embed*.

child available. This property allows us to determine visits graphs in a stable way. Some modifications are possible to this approach to schedule attributes less eagerly.

Further, for each nonterminal  $N$ , there must be a vertex in the visits graph with an empty configuration. An empty configuration consists of an empty configuration for the children of each production. In addition, for the root symbols of the grammar with a non-empty set of attributes, the configuration with all attributes defined must be part of the visits graph. The configurations of the children do not have to be fully defined. Also, there must be an appropriate edge between these configurations.

**Properties.** The invariants imposed on the visits graph ensure a number of properties for which we sketch proofs.

*The visits graph is acyclic.* The destination configuration connected by an edge is larger than the edge's source configuration, because the set of attributes of an edge is not empty. Edges thus connect distinct configurations. The configurations connected by edges form an ascending chain, and can thus not be cyclic.  $\square$

*The VPDGs are acyclic.* The VPDGs of the initial configurations are by definition acyclic. The transformation induced by a visit adds visit-vertices, although these keep the VPDG acyclic. A vertex that is available requires its (indirect) dependencies to be available. Thus, an available vertex cannot depend on an unavailable vertex. A visit-vertex connects available vertices to unavailable vertices of synthesized attributes of a child. To form a cycle, such an available vertex must depend (indirectly) on one of the synthesized attributes. Since such a vertex is unavailable, a cycle is not possible.  $\square$

#### 4 AGs with Commuting Rules

*Consistent visits to children.* The edges of the NDG of a child forms the edges between the child's inherited and synthesized attributes in the VPDG of a production that contains the child. Suppose that we visit child  $x$  with  $\bar{a}$  as the attributes of the visit. As induction hypothesis, we assume that the available inherited and synthesized attributes of  $x$  are part of a consistent visit sequence. To prove that the visit sequence with the visit added is consistent, the definition requires us to consider four cases. In each case, one of the attributes is an element of  $\bar{a}$ .

As the first case, given an attribute  $inh.x.y$  and an attribute  $syn.x.z$ , suppose that  $inh.y \leftarrow^+ syn.z$  in the NDG of  $x$ , and  $syn.x.z \in \bar{a}$ . Consequently, vertex  $syn.x.z$  is unavailable. Then  $inh.x.y \leftarrow syn.x.z$  in the VPDG, which means that  $inh.x.y$  is available. Hence,  $syn.x.z$  must be in the same or later visit as  $inh.x.y$ . Proofs for the remaining three cases are similar.  $\square$

*Consistent states.* When two potential edges in the visits graph converge, they result in the same state. Clearly, when two edges converge, the edges have an equivalent set of attributes  $\bar{a}_1$  as destination state, otherwise the edges would not converge. Then, for both a VPDG  $g_1$  of one edge, and a VPDG  $g_2$  for the same production of the other edge, the *reqs* sets for  $\bar{a}_1$  are equivalent. The attributes of a child's state are exactly those in the *reqs* set, hence the states of the children are equivalent.  $\square$

*Dependencies between attributes of the same kind.* A synthesized attribute  $syn.x.y$  can depend on a synthesized attribute  $syn.x.z$  and still be computed as part of the same visit when the inherited attributes both  $syn.x.y$  and  $syn.x.z$  depend on are available. Similarly, two inherited attributes can both be passed simultaneously to a child when they are both available. An inherited attribute that depends on a synthesized attribute of the same child, however, cannot be scheduled to the same visit.  $\square$

**Construction.** The size of the visits graph is in both the worst and average case exponential in the number of attributes. However, we normally need only a small portion of this graph. In the remainder of this chapter, we call the visits graph of a program a slice of the graph that we inferred from the program.

To incrementally construct this slice, we distinguish *partial* and final vertices and edges. A partial vertex contains the configuration  $\bar{a}$ , but not the administration for the productions. A final vertex does contain this information. Similarly, a partial vertex contains only the visit identifier  $i$  and the attributes  $\bar{a}$ .

The partial vertices and edges represent information of the graph that needs to be in the graph, but what we did not compute yet. The final vertices and edges represent already computed parts of the graph. For example, the initial vertices with an empty state are final vertices. For the root symbols, we insert a partial edge and partial destination vertex to the graph. As algorithm, we repeatedly take a partial edge with a final source vertex, and perform scheduling to turn it into a final edge, and consequently the destination vertex into a final vertex. Scheduling may result in new edges and vertices being added to the graph.

The algorithm that performs scheduling for final edges computes the information for the edge as a function of the source state and the attributes  $\bar{a}$  of the edge. This ensures that the



order in which we consider pending edges does not affect the resulting visits graph. Moreover, the visits graph is finite, thus if the computation for each pending edge is finite, then so is the whole computation.

As a given, the input to the algorithm is a set of attributes  $\bar{a}$  of the edge, and the VPDGs of the productions of the previous configuration. To compute the remaining administration of the edge, we infer the visits to the children. The *embed* relation tells us how. We first determine which vertices are in the *reqs* set. Then we repeatedly which of the attributes of the children are required and ready to be scheduled. Those form one group of simultaneous visits. For each visit, we possibly add a pending edge and a pending vertex to the visits graph, if such vertices are not yet in the graph. This process terminates because the VPDG is acyclic. With each iteration, there is either at least one vertex that can be scheduled, or we are done.

The compilation of the largest AG of UUAG takes less than a minute using a slightly optimized version of the algorithm as sketched above. To improve the performance, the construction of the graph is relatively straightforward to parallelize. The processing of each edge is independent and can be done in parallel. Most of the shared state is read-only. Only updates on the state need to be properly synchronized, but these updates happen relatively infrequent, and there is likely little contention. The construction of the graph is thus likely to scale very well. Also, the process may be done incrementally. When a change in the AG does not affect the PDGs of a nonterminal, the graphs constructed for that nonterminal so far may be reused as initial visits graph. However, the implementation is performant enough to be used in practice, even without such optimizations.

**Remarks.** Both the construction of the PDGs and the visits graph is a whole-program analysis of the sources related to an evaluation algorithm of a compiler for a particular tree (e.g. one stage in the compiler pipeline). The requirement that the sources related to one algorithm must be analyzed as a whole is usually not a problem in large compiler implementations, because individual algorithms are usually monolithic. When plugins are concerned, such plugins are typically a separate stage, and would thus be compiled independently. However, it is in theory possible to defer the computations of the visits graph to load-time, although then the composition of rules must also be determined at load-time, using meta programming facilities such as Template Haskell.

There are various customizations possible in the construction on the visits graph which may have severe consequences for the structure of the graph. The structure of the graph does not influence the outcome of the attribute evaluation, but may affect execution time.

At the moment of writing, we are still gathering empirical data regarding performance. In earlier measurements on UHC, the difference between on-demand (lazy) and eager (strict) attribute evaluation turned out to be insignificant. This may be due to the use of strictness annotations and DeepSeq in combination with the sequentialization of computations due to unifications on a chained substitution attribute. However, in earlier measurements with the AG implementation of the editor Proxima [Schrage and Jeurig, 2004], execution time almost halved when eager AG evaluation was used. Note that with the rise of multi-core computing, the effects of visit sequences on parallel behavior may still be significant [Kuiper and Swierstra, 1990, Klaiber and Gokhale, 1992, Wang and Ye, 1991].

## 4.6 Optimizations

The shape of the graph may have an impact on the performance of the generated code. In our approach, visits are likely to be small and independent, which is beneficial for parallelism and incremental evaluation. However, when the visits graph is huge, code size becomes a more pressing issue. The visits graph of the largest AG in the UHC project features about 10,000 configurations. Hence, we require measures to limit the size of this graph.

**Subsumption.** When the graph is huge, there are many contexts that are similar, yet differ in one or more attributes because such an attribute was not needed or only needed later. Such a small difference can easily cause many visits to be needed in the graph.

An edge  $i$  from a configuration  $\bar{a}$  with attributes  $\overline{inh.y}$  and  $\overline{syn.z}$  subsumes an edge  $k$  from  $\bar{a}$  with attributes  $\overline{inh.p}$  and  $\overline{syn.q}$  when  $\overline{syn.y} \subseteq \overline{inh.p}$  and  $\overline{syn.q} \subseteq \overline{syn.z}$ . When we are about to declare a visit, but it is subsumed by an already declared visit, we may use that visit instead. This approach can potentially save many contexts, at the expense of producing some results that are not needed yet. That is usually not a problem, because non-trivial computations of some node normally have dedicated inherited attributes, thus if one of these inherited attributes is needed then so is the non-trivial computation that computes the synthesized attribute.

A downside of the subsumption approach is that the order in which edges in the visits graph are considered may influence the outcome. The effects of such approaches requires further investigation.

We experimented with the strategy to determine the inherited attributes of a visit based on the synthesized attributes that are required, but determining the largest set of synthesized attributes that can be computed from the inherited attributes available so far. This strategy reduced the graph of UHC's largest AG to 1,500 nodes, at the slight expensive of returning one or two attributes more than is strictly necessary. Under the assumption that complex synthesized attributes are always dependent on their own set of inherited attributes, the additional cost is negligible.

**Partial Kastens.** Two other decisive factors are the number of attributes and productions, and sparse dependencies between attributes. The number of productions is typically fixed, but the number of attributes and the dependencies can be influenced. We can produce an algorithm for any consistent visit sequence. Consequently, a PDGs can be replaced with supergraphs as long as these remain acyclic.

In our experience, orderable AGs are absolutely non-circular (Section 1.3.4). If for one production there is a child  $x$  with nonterminal  $N$  that has a dependency of an attribute  $inh.x.y$  on an attribute  $syn.x.z$ , then this dependency can also be imposed for all other children with nonterminal  $N$ . For such an AG, the approach of Kastens [1980] is applicable.

The approach of Kastens determines a total order on the attributes in the NDGs using a late-as-possible strategy. As mentioned earlier, this approach likely causes the graphs to become cyclic. By removing the edges between vertices of strongly connected components that are not in the original PDG, we obtain a non-cyclic supergraph of the original PDG. This

supergraph has Kastens' algorithm partially applied, and is likely to have a significant lower number of possible contexts.

For large AGs, this appears to be the most effective step to limit the state explosion caused by many productions and attributes. It does not require manual intervention. On the other hand, it restricts the freedom in choosing smaller and independent visits.

**Attribute elimination.** In combination with copy rules, it is common practice to define attributes on nonterminals where they are actually unused, or only used in chains of copy rules. During the development of an AG, such attributes also show up because the AG is not finished, thus not all attributes are in use.

Superfluous attributes seem innocent, but actually make the scheduling harder. These attributes and their rules for such nonterminals are largely independent, thus easily lead to many contexts in the visits graph, because many ways to interleave them are allowed. Further, the Kastens' algorithm typically schedules them too early, which causes cycles in the dependency graphs, thus makes the above approach to reduce contexts less effective.

A combination of dead-code elimination and copy propagation [Nielson et al., 1999] can be used to eliminate superfluous attributes of a nonterminal. As an additional benefit, their removal may improve the performance of the application, because it prevents the trivial copying of attributes around the tree.

Similarly to the dependency analysis itself, many analyses for AGs can be specified as a recursive set of constraints. The common pattern is that some property of a nonterminal is determined by combining the properties of each production, which we call abstraction. This property of the nonterminal is then instantiated for each a child of the nonterminal. A fixpoint can then be computed starting with a fixed value for the nonterminals that are roots, and a bottom value for the other nonterminals.

In a similar way as shown in Section 4.4, we can define that an attribute is *live* if it is a dependency of a live attribute. For the root nonterminal, all synthesized attributes are live. Since this analysis is defined in terms of the attribute dependencies, the analysis can straightforwardly be defined in terms of the dependency graphs. As with dead-code elimination, the attributes and rules that are not live are removed from the grammar.

Similarly, a collection attribute [Magnusson et al., 2007] is *empty* if it is composed with either a copy rule or a monoid's append from empty attributes. In this case, we start with non-empty as bottom value, and do not treat start symbols of the grammar in a special way. Instead, a use rule that depends on no attributes becomes empty as value eventually.

Finally, the output attribute of a copy rule is a *copy* of an attribute  $x$ , where  $x$  either equals  $z$  when the input attribute  $y$  is a copy of another attribute  $z$ , or  $x$  equals  $y$  otherwise. This also applies to rules for which we can syntactically determine that its body is essentially the identity function, For other rules, the output attribute is not a copy of another attribute. Initially, each attribute is not a copy of any attribute. set of attributes. During abstraction, a synthesized attribute may be a copy of an inherited attribute if all productions agree on that attribute.

Given the results of the copy and empty analyses, a rule that depends on an attribute  $x$  that is a copy of  $y$  can substitute  $y$  for  $x$ . A rule that depends on an empty attribute can substitute the attribute with the empty value, and thus remove the dependency on the attribute. Finally,

## 4 AGs with Commuting Rules

dead-code elimination cleans up the unused copies, unused empty attributes, and unused other attributes.

The updatable attributes provide some more options for extensions. When an attribute is *constant* or the attribute is *unique* [Hage et al., 2007], these attributes can be stored as a mutable structure in a global state, and replaced by a single attribute that stores a reference to that structure. These are analyses that are relatively straightforward to implement and exploit for BarrierAG, in contrast to general purpose programming languages, which have complications due to data types and higher-order functions.

## 4.7 Execution Plans and Generated Code

With each vertex in the visits graph, we uniquely associate an identifier  $j$ . For each edge of the visits graph, and for each of its productions, we construct an execution plan. The vertices of the VDPG that are required for the new state, but not required for the old state, are the edges that belong in the plan. In Chapter 3, we described how to derive a total order for the vertices. In this section, we simply assume that we take a topological sort of the graph.

**Plan representation.** Figure 4.19 shows the syntax of execution plans, and the execution plans for the production *Bin*. We organize the plans per production. For each production, it contains an execution plan for each edge in the visits graph. The rules are ordered, and visits to children are made explicit.

**Definition** (Intra-visit dependency). With each vertex in the visits graph, we associate a set of *intra-visit* dependencies. These are values (denoted as descriptors  $d$ ) in the state of a node that are potentially needed in later visits. Given a visits graph  $V$ , the set of descriptors contained in  $s$  for a production  $P$  is *intra*  $P$   $s$  in Figure 4.20 (explained below).

The set of descriptors is determined by taking the descriptors introduced by a visit, and those needed by rules of later visits. Also note that the graph  $g$  is the graph of the new state for *defs*, and the graph of the old state for *uses*. The difference is in the state of the child-nodes. Finally, the set of intra-visit dependencies of the initial state and each final state is empty.

From execution plans, we generate visit functions (see also Section 1.3.5). The process for individual visit functions is largely conventional (Section 2.1, and Section 5.3). However, the weaving of the visit functions is more complex, because we encode the visits graph, which is in general not a linear sequence of visits.

The visits graph specifies the possible states of an attributed tree, and models the visits that can be done on it. Moreover, it specifies which attributes are in which state, and which attributes are an input or output of a visit. In the generated code, an attributed tree is a value that represents a vertex in the visits graph. A visit function represents an edge. Figure 4.21 shows their types for the example. Below, we explain these types.

**Types.** For each vertex  $j$  of a nonterminal  $N$  in the visits graph, we generate a type  $T\_N\_sj$ . This is the type for an attributed tree with nonterminal  $N$  and in configuration  $j$ . For each edge  $i$  of a nonterminal  $N$  in the visits graph, we generate a type  $T\_N\_vi$ . This is the type

$S ::= \mathbf{sem} \Gamma P : N \bar{c} \bar{q} j$  -- execution plans  $\bar{q}$  of production  $P$ , and initial config  $j$   
 $c ::= \mathbf{conf} j \bar{a}$  -- description of a configuration  $j$   
 $q ::= s ; s' ; v$  -- edge between configurations  $s$  and  $s'$   
 $v ::= \mathbf{visit} i : N \bar{a} \bar{r}$  -- visit combined with rules of the visit  
 $r ::= \mathbf{child} x : N = f \bar{z}^{\triangleright}$  -- (higher order) child declaration  
    |  $x : p \bar{z}^{\triangleleft} = f \bar{z}^{\triangleright}$  -- evaluation rule  
    |  $\mathbf{sim} \bar{m}$  -- simultaneous invocations  
 $m ::= \mathbf{invoke} x i : N$  -- child invocation  
 $s ::= \mathbf{conf} j \bar{k}$  -- configuration of a node  
 $k ::= \mathbf{child} x : N j$  -- configuration of child  $x$   
 $j$  -- configuration identifier

```

sem { $f_1 = \mathit{field}.l, f_2 = \mathit{field}.r, f_3 = \mathit{min}, f_4 = \mathit{Bin}$ }  $\mathit{Bin} : \mathit{Tree} 1$ 
  { conf 1  $\emptyset$ , conf 2 { $\mathit{syn.gath}$ }, conf 3 { $\mathit{inh.mini}, \mathit{syn.gath}, \mathit{syn.repl}$ } }
  [ conf 1 { child  $l : \mathit{Tree} 1, \mathbf{child} r : \mathit{Tree} 1$  }
    ; conf 2 { child  $l : \mathit{Tree} 2, \mathbf{child} r : \mathit{Tree} 2$  }
    ; visit 1 :  $\mathit{Tree}$  child  $l : \mathit{Tree} = f_1$ 
      child  $r : \mathit{Tree} = f_2$ 
      sim { invoke  $l 1 : \mathit{Tree}, \mathbf{invoke} r 1 : \mathit{Tree}$  }
         $r_3 : \mathit{id} \mathit{syn.lhs.gath}^\bullet = f_3 \mathit{syn.l.gath}^\bullet \mathit{syn.r.gath}^\bullet$ 
    , conf 2 { child  $l : \mathit{Tree} 2, \mathbf{child} r : \mathit{Tree} 2$  }
    ; conf 3 { child  $l : \mathit{Tree} 3, \mathbf{child} r : \mathit{Tree} 3$  }
    ; visit 2 :  $\mathit{Tree}$   $r_5 : \mathit{id} \mathit{inh.l.mini}^\bullet = \mathit{id} \mathit{syn.r.gath}^\bullet$ 
       $r_6 : \mathit{id} \mathit{inh.r.mini}^\bullet = \mathit{id} \mathit{syn.l.gath}^\bullet$ 
      sim { invoke  $l 2 : \mathit{Tree}, \mathbf{invoke} r 2 : \mathit{Tree}$  }
         $r_4 : \mathit{id} \mathit{syn.lhs.repl}^\bullet = f_4 \mathit{syn.l.repl}^\bullet \mathit{syn.r.repl}^\bullet$ 
    , conf 1 { child  $l : \mathit{Tree} 1, \mathbf{child} r : \mathit{Tree} 1$  }
    ; conf 3 { child  $l : \mathit{Tree} 3, \mathbf{child} r : \mathit{Tree} 3$  }
    ; visit 3 :  $\mathit{Tree}$  child  $l : \mathit{Tree} = f_1$ 
      child  $r : \mathit{Tree} = f_2$ 
      sim { invoke  $l 1 : \mathit{Tree}, \mathbf{invoke} r 1 : \mathit{Tree}$  }
         $r_3 : \mathit{id} \mathit{syn.lhs.gath}^\bullet = f_3 \mathit{syn.l.gath}^\bullet \mathit{syn.r.gath}^\bullet$ 
         $r_5 : \mathit{id} \mathit{inh.l.mini}^\bullet = \mathit{id} \mathit{syn.r.gath}^\bullet$ 
         $r_6 : \mathit{id} \mathit{inh.r.mini}^\bullet = \mathit{id} \mathit{syn.l.gath}^\bullet$ 
      sim { invoke  $l 2 : \mathit{Tree}, \mathbf{invoke} r 2 : \mathit{Tree}$  }
         $r_4 : \mathit{id} \mathit{syn.lhs.repl}^\bullet = f_4 \mathit{syn.l.repl}^\bullet \mathit{syn.r.repl}^\bullet$ 

```

Figure 4.19: Syntax of execution plans, and the plans of production  $\mathit{Bin}$ .

#### 4 AGs with Commuting Rules

$$\begin{aligned}
\text{intra } P s &= \cup \{ (\text{uses } P s s' \cup \text{intra } P s') - \text{defs } P s s' \mid (s; s'; v) \in V \} \\
\text{uses } P \langle \mathbf{conf } N \bar{a}_0 \bar{p}_0 \rangle \langle \mathbf{conf } N \bar{a}_1 \bar{p}_1 \rangle \mid \langle \mathbf{prod } P \bar{k} g \rangle \in \bar{p}_0 &= \\
&\cup (\text{map } (\text{uses } P) (\text{reqs } \bar{a}_1 g - \text{reqs } \bar{a}_0 g)) \\
\text{uses } P \langle k.x.y \rangle &= \emptyset \\
\text{uses } P \langle \mathbf{child } x \bar{a} \rangle &= \cup \{ \text{map } \text{uses } \bar{z} \mid \langle \mathbf{child } x : N = f \bar{z}^{\bar{p}} \rangle \in \text{rules } P \} \\
&\mid \text{length } \bar{a} > 0 = \{ \mathbf{child } x \bar{a} \} \\
\text{uses } P \langle \mathbf{rule } x \rangle &= \cup \{ \bar{z}_2 \cup \text{map } \text{uses } \bar{z}_1^{\bar{q}} \mid \langle x : p \bar{z}_1^{\bar{q}} = f \bar{z}_2^{\bar{p}} \rangle \in \text{rules } P \} \\
\text{uses } \langle z^\circ \rangle &= \{ z \} \\
\text{uses } \langle z^\bullet \rangle &= \emptyset \\
\text{uses } \langle z^\times \rangle &= \emptyset \\
\text{defs } P \langle \mathbf{conf } N \bar{a}_0 \bar{p}_0 \rangle \langle \mathbf{conf } N \bar{a}_1 \bar{p}_1 \rangle \mid \langle \mathbf{prod } P \bar{k} g \rangle \in \bar{p}_1 &= \\
&\cup (\text{map } (\text{defs } P) (\text{reqs } \bar{a}_1 g - \text{reqs } \bar{a}_0 g)) \\
\text{defs } P \langle k.x.y \rangle &= \emptyset \\
\text{defs } P \langle \mathbf{child } x \bar{a} \rangle &= \{ \mathbf{child } x \bar{a} \} \\
\text{defs } P \langle \mathbf{rule } x \rangle &= \cup \{ \text{map } \text{defs } \bar{z}_1^{\bar{q}} \mid \langle x : p \bar{z}_1^{\bar{q}} = f \bar{z}_2^{\bar{p}} \rangle \in \text{rules } P \} \\
\text{defs } \langle z^\circ \rangle &= \emptyset \\
\text{defs } \langle z^\bullet \rangle &= \{ z \} \\
\text{defs } \langle z^\times \rangle &= \{ z \}
\end{aligned}$$

**Figure 4.20:** Intra-visit dependencies of nodes.

for a visit function that takes the tree from its source configuration  $T\_N\_s \llbracket i \rrbracket_{\text{source}}$  to its target destination  $T\_N\_s \llbracket i \rrbracket_{\text{target}}$ .

We can apply one operation on an attributed tree. Given an *typed key*  $K\_N\_s_j t$  and a tree  $T\_N\_s_j$ , the function  $\text{inv } N\_s_j :: T\_N\_s_j \rightarrow K\_N\_s_j t \rightarrow t$  provides us with the visit function of type  $t$ :

**data**  $T\_N\_s_j$  **where**  $C\_N\_s_j :: \{ \text{inv } N\_s_j :: \forall t. K\_N\_s_j t \rightarrow t \} \rightarrow T\_N\_s_j$

The constructor  $C\_N\_s_j$  is essentially a wrapper around the *inv* function.

The type  $t$  can be chosen by a parent by providing a key with this type. The key  $K\_N\_s_j t$  is a type index. The child can inspect the key to discover which type is actually represented by  $t$ :

**data**  $K\_N\_s_j t$  **where**

$K\_N\_s_j v_{i_1} :: K\_N\_s_j T\_N\_s_j v_{i_1}$

...

$K\_N\_s_j v_{i_n} :: K\_N\_s_j T\_N\_s_j v_{i_n}$

For each outgoing edge  $i$  of  $j$ ,  $K\_N\_s_j$  contains a key  $K\_N\_s_j v_i$  that serves as evidence that  $t$

```

type  $T\_Tree = T\_Tree_{s_1}$  -- initial configuration of the tree
    -- type of tree in a given state  $s$  (function from key to visit)
data  $T\_Tree_{s_1}$  where  $C\_Tree_{s_1} :: \{ inv\_Tree_{s_1} :: \forall t. K\_Tree_{s_1} t \rightarrow t \} \rightarrow T\_Tree_{s_1}$ 
data  $T\_Tree_{s_2}$  where  $C\_Tree_{s_2} :: \{ inv\_Tree_{s_2} :: \forall t. K\_Tree_{s_2} t \rightarrow t \} \rightarrow T\_Tree_{s_2}$ 
data  $T\_Tree_{s_3}$  where  $C\_Tree_{s_3} :: \{ inv\_Tree_{s_3} :: \forall t. K\_Tree_{s_3} t \rightarrow t \} \rightarrow T\_Tree_{s_3}$ 
    -- type of a key, which identifies a visit  $v$  from state  $s$ 
data  $K\_Tree_{s_1} t$  where
     $K\_Tree_{v_1} :: K\_Tree_{s_1} T\_Tree_{v_1}$ 
     $K\_Tree_{v_3} :: K\_Tree_{s_1} T\_Tree_{v_3}$ 
data  $K\_Tree_{s_2} t$  where
     $K\_Tree_{v_2} :: K\_Tree_{s_2} T\_Tree_{v_2}$ 
data  $K\_Tree_{s_3} t$  where -- empty data declaration
    -- type of a visit  $v$ , with continuation as the new state  $s$ 
type  $T\_Tree_{v_1} = IO (Int, T\_Tree_{s_2})$ 
type  $T\_Tree_{v_2} = Int \rightarrow IO (Tree, T\_Tree_{s_3})$ 
type  $T\_Tree_{v_3} = Int \rightarrow IO (Int, Tree, T\_Tree_{s_3})$ 

```

**Figure 4.21:** Types of keys and semantic functions.

equals  $T\_N_{v_i}$ . The type  $K\_N_{s_j}$  has no constructors when  $j$  has no outgoing edges in the visits graph. Indeed, a tree in such a configuration cannot be visited.

The type of a visit function  $T\_N_{v_i}$  is a function of values of the visit's inherited attributes to a computation of a tuple of values of the visit's synthesized attributes, and the new state of the tree. Since BarrierAG includes updatable attributes, the computation takes place in the IO monad:

$$\mathbf{type} \ T\_N_{v_i} = \tau_{inh_1} \rightarrow \dots \rightarrow \tau_{inh_n} \rightarrow IO (\tau_{syn_1}, \dots, \tau_{syn_m}, T\_N_{s[[i]]_{target}})$$

The type  $\tau_{inh_k}$  is the type declared for the inherited attribute  $inh_k$  of nonterminal  $N$ , and  $\tau_{syn_k}$  the type for the synthesized attribute  $syn_k$ .

**Translation of semantics-blocks.** Figure 4.22 gives the generated code for a production<sup>2</sup>. A sem-block is translated to a *node constructor* function  $st_j$  for each state  $j$ , which given values for the intra-dependencies of  $j$ , returns a node with this state, thus of type  $T\_N_{s_j}$ . The node constructor  $st_1$  constructs the initial state. Therefore it has an empty set of intra-dependencies and is thus represents the initial value of the node. In contrast, the node constructor  $st_2$  takes the states of the live children as parameter, and values of the live attributes.

<sup>2</sup> The full code of the example can be downloaded from: <https://svn.science.uu.nl/repos/project.ruler.papers/archive/ExampleWarren.hs>. It is compilable with GHC version 6.12.3.

#### 4 AGs with Commuting Rules

```

sem_Bin :: T_Tree → T_Tree → T_Tree
sem_Bin field_l field_r = st1 where
  st1 = let k1 :: K_Tree_s1 t → t
        k1 K_Tree_v1 = v1
        k1 K_Tree_v3 = v3
        k1 _         = error "unreachable"
        v1 :: T_Tree_v1
        v1         = do l1 ← return f1
                       r1 ← return f2
                       (l_gath, l2) ← inv_Tree_s1 l1 K_Tree_v1
                       (r_gath, r2) ← inv_Tree_s1 r1 K_Tree_v1
                       lhs_gath ← return.id $f3 l_gath r_gath
                       return (lhs_gath, st2 l2 r2 l_gath r_gath)

        v3 :: T_Tree_v3
        v3 lhs_mini = do l1 ← return f1
                       r1 ← return f2
                       (l_gath, l2) ← inv_Tree_s1 l1 K_Tree_v1
                       (r_gath, r2) ← inv_Tree_s1 r1 K_Tree_v1
                       lhs_gath ← return.id $f3 l_gath r_gath
                       l_mini ← return.id $id r_gath
                       r_mini ← return.id $id l_gath
                       (l_repl, l3) ← inv_Tree_s2 l2 K_Tree_v2 l_mini
                       (r_repl, r3) ← inv_Tree_s2 r2 K_Tree_v2 r_mini
                       lhs_repl ← return.id $f4 l_repl r_repl
                       return (lhs_gath, lhs_repl, st3)

    in C_Tree_s1 k1
  st2 l2 r2 l_gath r_gath
    = let k2 :: K_Tree_s2 t → t
        k2 K_Tree_v2 = v2
        k2 _         = error "unreachable"
        v2 :: T_Tree_v2
        v2 lhs_mini = do l_mini ← return.id $id r_gath
                       r_mini ← return.id $id l_gath
                       (l_repl, l3) ← inv_Tree_s2 l2 K_Tree_v2 l_mini
                       (r_repl, r3) ← inv_Tree_s2 r2 K_Tree_v2 r_mini
                       lhs_repl ← return.id $f4 l_repl r_repl
                       return (lhs_repl, st3)

    in C_Tree_s2 k2
  st3 = let k3 :: K_Tree_s3 t → t
        k3 _         = error "unreachable"
    in C_Tree_s3 k3
  f1 = field_l ; f2 = field_r ; f3 = min ; f4 = Bin

```

**Figure 4.22:** Generated code of the production *Bin*.



$\llbracket \text{child } x : N = f \overline{z^{\triangleright}} \rrbracket_{\text{gen}}$	$=$	$\llbracket x \rrbracket_{\text{gen}} (\text{init } N) \leftarrow \text{lift}_{ \overline{z^{\triangleright}} } f \llbracket \overline{z^{\triangleright}} \rrbracket_{\triangleright}$
$\llbracket x : p \overline{z^{\triangleleft}} = f \overline{z^{\triangleright}} \rrbracket_{\text{gen}}$	$=$	$\overline{y} \leftarrow \text{lift}_{ \overline{z^{\triangleright}} } f \llbracket \overline{z^{\triangleright}} \rrbracket_{\triangleright}$ $\llbracket \overline{z^{\triangleleft}} \rrbracket_{\triangleleft y}$ <b>where</b> $\overline{y}$ <i>fresh</i>
$\llbracket \text{invoke } x \ i : N \rrbracket_{\text{gen}}$	$=$	<b>let</b> $\text{vis} = \text{invoke}_{s_{[i]_{\text{source}}}} \llbracket x \rrbracket_{\text{source}} (i) (K\_N\_v_i)$ $(\llbracket x \rrbracket_{\text{syn}} (i), \llbracket x \rrbracket_{\text{target}} (i)) \leftarrow \text{vis } \llbracket x \rrbracket_{\text{inh}} (i)$
$\llbracket x \rrbracket_{\text{inh}} (i)$	$=$	$\{ \llbracket \text{inh}.x.y \rrbracket_{\text{gen}} \mid \text{inh}.y \leftarrow a_{N_i} \}$
$\llbracket x \rrbracket_{\text{syn}} (i)$	$=$	$\{ \llbracket \text{syn}.x.y \rrbracket_{\text{gen}} \mid \text{syn}.y \leftarrow a_{N_i} \}$
$\llbracket \overline{z^{\bullet}} \rrbracket_{\triangleleft y}$	$=$	<b>let</b> $\llbracket z \rrbracket_{\text{gen}} = y$
$\llbracket \overline{z^{\circ}} \rrbracket_{\triangleleft y}$	$=$	$\text{writeIORef } \llbracket z \rrbracket_{\text{gen}} y$
$\llbracket \overline{z^{\times}} \rrbracket_{\triangleleft y}$	$=$	$\llbracket z \rrbracket_{\text{gen}} \leftarrow \text{newIORef } y$
$\llbracket \overline{z^{\bullet}} \rrbracket_{\triangleright}$	$=$	$\text{return } \llbracket z \rrbracket_{\text{gen}}$
$\llbracket \overline{z^{\circ}} \rrbracket_{\triangleright}$	$=$	$\text{readIORef } \llbracket z \rrbracket_{\text{gen}}$
$\llbracket h.c.x \rrbracket_{\text{gen}}$	$=$	$h\_c\_x$
$\llbracket x \rrbracket_{\text{gen}} (j)$	$=$	$x\_j$

**Figure 4.23:** Translation scheme for rules in the execution plan.

The body of a node constructor is a wrapper around a function  $k_j$ , which returns the visit function  $v_i$  given the key  $K\_N\_v_i$  that identifies one of the outgoing edges of  $j$ . For each visit  $i$  that is a successor of configuration  $j$ , we generate a visit function  $v_i$ . The visit function  $v_i$  takes values for the inherited attributes as parameter that are declared on edge  $i$ . It returns a computation that gives a tuple of the synthesized attributes that are declared on edge  $i$ . In addition, it returns the result state of the node by applying the constructor for the next state to the values of that state's intra dependencies.

Figure 4.23 shows a straightforward translation of the rules in the execution plan. We assume that barrier attributes and dependency rules are stripped from the execution plan and the administration after the visits graph has been constructed.

For each visit to a child we pass as additional parameter the appropriate key to the invoke-function of the child's state, which results in the visit function  $\text{vis}$ . The function  $\text{vis}$  subsequently takes the inherited attributes as parameters.

For an attribute occurrence  $z$  at an input position, either the transcribed identifier  $\llbracket z \rrbracket_{\text{gen}}$  is  $z$ 's value, or is a reference that can be read from to provide  $z$ 's value. In a similar way, an attribute occurrence  $z$  at an output position is either stored as the transcribed identifier  $\llbracket z \rrbracket_{\text{gen}}$ , or written to the reference under that name.

**Remarks.** A nice property of the translation is that we explicitly declare the types of the visit functions with type signatures. Since the types of the functions  $f$  are monomorphic, type errors are usually to be reported in functions  $f$ , which are defined in the actual source code, so that type errors can be related back to the original locations in the source file. We do not

need to know the types of local attributes. Also, when we allow type variables in the types, then these can be supported in our scheme using scoped type variables.

The translation can be optimized in various ways. When a configuration does not have outgoing edges, a visit to a child that ends in this configuration does not need to construct nor return the new state of the child, as the child cannot be visited anymore, and the state is actually empty according to the definition of *intra*. When a configuration has one outgoing edge, then the selection of a visit via a key is not needed. Also, when the visits graph for a nonterminal is a tree, a single key that identifies the path in the tree can be given to the child when it is created, instead of one segment of the path for each visit.

Instead of relying on Haskell to construct closures, the node constructors can use an untyped, updatable array instead. With conventional techniques from register scheduling, the state can be represented such that needless copying of states is avoided during visits, which makes visits cheap. In this thesis, we do not venture down this path, and rely on Haskell to handle closures. A compiler that employs uniqueness and usage analysis [de Vries et al., 2007, Hage et al., 2007] can apply this optimization transparently.

When a simultaneous invocation group contains more than one visit, we can use *forkIO* to allow the Haskell runtime to evaluate the visits in parallel, since these do not have common dependencies.

## 4.8 Generalization to Phases

We use `BarrierAG` as a host language to describe phases. A nonterminal declares a set of phases. Attributes may be associated uniquely to a phase. A phase corresponds to one or more implicit visits. Since a visit describes the smallest unit of evaluation of a node in the tree, a phase describes a larger unit of evaluation. It allows us to express properties of chunks of AG evaluation, without resorting to the low level details of visits.

The following example is a possible declaration of phases for nonterminal *Tree*. The indentation determines which attribute is declared in which phase. The scope of the phase declaration ends before a keyword at the same indentation level:

```

itf Tree                -- declaration of attributes and phases of Tree
  syn gath :: Int       -- attribute not assigned to a phase
  phase distribute      -- declaration of a phase distribute
    inh mini :: Int    -- attribute mini in phase distribute
  phase transform      -- declaration of a phase transform
    syn repl :: Tree  -- attribute repl in phase transform

```

These phase declarations are unordered, and can be specified in any order.

Figure 4.24 shows the notation for phase interfaces. There is a high similarity with the notation for visit interfaces of Chapter 3. The main difference is that we may specify multiple blocks of subsequent phases in a phase-block. The lexical nesting of phase declarations (on a nonterminal) and phase blocks (in a sem-block) impose a partial order on phases. Also order-rules in combination with **begin** *lhs*  $\rho$  and **end** *lhs*  $\rho$  imposes a partial order. A phase-block

$I ::= \mathbf{itf} N \bar{q}$	-- a set of declaration of a phase-interface
$q ::= a$	-- toplevel attribute decl (not associated with a phase)
$\mathbf{phase} \rho \bar{q}$	-- phase with attribute declarations
$d ::= \dots$	-- descriptors extended with phases
$\mathbf{begin} x \rho$	-- begin of a phase $\rho$ of child $x$
$\mathbf{end} x \rho$	-- end of a phase $\rho$ of child $x$
$s ::= \mathbf{sem} N \mathbf{prod} P \bar{r} \bar{t}$	-- common rules $r$ and a set of phase blocks $\bar{t}$
$t ::= \mathbf{phase} \rho \bar{r} \bar{t}$	-- common rules $r$ and subsequent phases $\bar{t}$
$r ::= \dots$	-- AG rules
$\mathbf{invoke} \rho \mathbf{of} \bar{c} z$	-- specifies invocation of phase $\rho$ of $\bar{c}$ with strategy $z$

**Figure 4.24:** Notation for phase interfaces.

of phase  $\rho_2$  that is nested in a phase block  $\rho_1$  is evaluated either during the evaluation of  $\rho_1$  when there is a constraint  $\mathbf{begin} lhs \rho_2 \prec \mathbf{end} lhs \rho_1$ , or after the evaluation of  $\rho$  otherwise.

Invoke-rules may be explicitly given or be implicit. An invoke-rule  $r$  with a phase  $\rho$  corresponds to one or more actual visits to a child  $x$ . The rule itself precedes the first of these visits, thus  $r \prec \mathbf{begin} x \rho$ .

The lexical scope of a phase block, which is relevant for local attributes and default-rules, is only determined by the nesting of phase blocks in a production. Possibly more constraints on the order induced by other productions or order-rules are not taken into account for scoping. This technical detail ensures that we can determine the vertices of the PDGs before the dependency analysis takes place.

We essentially have two main evaluation algorithms to choose from: demand-driven evaluation and statically ordered evaluation. We may specify several properties of a phase, such as cyclic or acyclic, and pure or impure. Not all combinations are possible. For example, a cyclic phase must use on-demand evaluation, and may not be impure. Also, the properties impose constraints on rules in such a phase, or on rules that invoke such a phase. For example, an invoke-rule in a pure phase may not invoke an impure phase on a child. The scheduling of rules takes such constraints into account (Section 3.5.2). In a host language with lazy evaluation, the on-demand algorithm is a simplification of the eager algorithm, hence in this thesis we focus only on the latter.

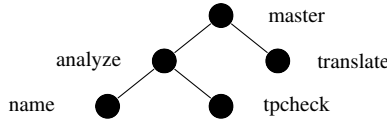
**Foundation.** We express phases in terms of BarrierAG, which we sketch in Figure 4.25. A dependency rule  $k.x \prec k.y$  on attr-blocks of  $N$  is syntactic sugar for rule  $k.lhs.x \prec k.lhs.y$  in each production of  $N$ . For each phase  $\rho$  of nonterminal  $N$ , we introduce two barrier-attributes  $begin_\rho$  and  $end_\rho$ . The attributes of the phase are enclosed by these barriers. There is one master phase  $N$  for each nonterminal  $N$  where all attributes and phases are enclosed by via dependencies on its barriers.

The lexical nesting of phases and rules induces additional order constraints, as sketched by

#### 4 AGs with Commuting Rules

<b>attr</b> $N$	<b>inh</b> $begin_N$	<b>barrier</b>	-- begin master phase for $N$
	<b>syn</b> $end_N$	<b>barrier</b>	-- end master phase for $N$
<b>attr</b> $N$	<b>inh</b> $begin_\rho$	<b>barrier</b>	-- begin barrier for each phase $\rho$ of $N$
	<b>syn</b> $end_\rho$	<b>barrier</b>	-- end barrier for each phase $\rho$ of $N$
<b>attr</b> $N$	$inh.begin_\rho \prec syn.end_\rho$		-- begin before end for each phase $\rho$
	$inh.begin_N \prec inh.begin_\rho$		-- begin $\rho$ after begin master phase
	$syn.end_\rho \prec syn.end_N$		-- end $\rho$ before end master phase
<b>attr</b> $N$	$inh.begin_\rho \prec k.y$		-- for each attribute $k.y$ of phase $\rho$
	$k.z \prec syn.end_\rho$		-- for each attribute $k.y$ of phase $\rho$
	$inh.begin_\rho \prec inh.begin_{\rho'}$		-- for each nested phase $\rho'$
	$syn.end_{\rho'} \prec syn.end_\rho$		-- for each nested phase $\rho'$
	$inh.begin_N \prec k.y$		-- for each attribute $k.y$ not in a phase
	$k.z \prec syn.end_N$		-- for each attribute $k.y$ not in a phase

**Figure 4.25:** Sketch of a translation of phases to BarrierAG.



**Figure 4.26:** Phase nesting visualized as a tree.

the following example:

<b>sem</b> $N \bar{r}_1$	-- $inh.lhs.begin_N \prec \bar{r}_1$
<b>phase</b> $\rho_1$	-- $inh.lhs.begin_N \prec inh.lhs.begin_{\rho_1}$
$\bar{r}_2$	-- $inh.lhs.begin_{\rho_1} \prec \bar{r}_1$
<b>phase</b> $\rho_2$	-- $inh.lhs.begin_{\rho_1} \prec inh.lhs.begin_{\rho_2}$
$\bar{r}_3$	-- $inh.lhs.begin_{\rho_2} \prec \bar{r}_3$

With order-dependencies, and with syntax as demonstrated in Chapter 3, more dependencies between phases may be specified.

Further, we take the union of all constraints on phases of nonterminal  $N$  from each child with nonterminal  $N$ , and integrate these in the NDG of  $N$ . The PDGs must remain acyclic, otherwise the constraints on phases are inconsistent, which is considered a static error.

**Phase nesting.** A phase may be contained inside another phase. The parent of a node can invoke the contained phase of the node as part of the evaluation of the containing phase. This

model introduces levels of granularity that allow for more concise specifications. We can visualize this model as a tree, which we show in Figure 4.26.

**Definition** (Nesting tree). A *nesting tree* describes the nesting of phase declarations.

In a nesting tree, the nodes are phases. When a node  $\rho_2$  is a child of a node  $\rho_1$ ,  $\rho_1$  is a nested phase of  $\rho_2$ . The above exemplary tree corresponds to the following phase interface:

<b>itf</b> <i>Expr</i>	-- defines the siblings for the root (master phase)
<b>phase</b> <i>analyze</i>	-- nested phase in master phase
<b>phase</b> <i>name</i>	-- nested phase in phase <i>name</i>
<b>phase</b> <i>tpcheck</i>	-- nested phase in phase <i>tpcheck</i>
<b>phase</b> <i>translate</i>	-- nested phase in master phase

The above phase interface describes that during the evaluation of the *analyze* phase of a child, the name phase of that child is invoked. Different strategies may be specified for the name phase than for the *analyze* phase.

The nesting tree can be inferred from the NDG. Given a nonterminal  $N$ ,  $N$ 's phase  $\rho_1$  is a child of  $N$ 's phase  $\rho_2$ , if either:

- $inh.begin_{\rho_2} \leftarrow^+ inh.begin_{\rho_1}$  and  $syn.end_{\rho_1} \leftarrow^+ syn.end_{\rho_2}$ . In this case,  $\rho_1$  is fully enclosed by  $\rho_2$ .
- there exists a  $\rho_3$  so that  $\rho_3$  is a child of  $\rho_1$ , and  $\rho_3$  is a child of  $\rho_2$ , with in the NDG  $inh.begin_{\rho_2} \leftarrow^+ inh.begin_{\rho_1}$  or  $syn.end_{\rho_1} \leftarrow^+ syn.end_{\rho_2}$ . In this case,  $\rho_3$  is both a child of  $\rho_1$  and  $\rho_2$ , which we resolve by making  $\rho_1$  a child of  $\rho_2$ .

The constraints form a directed graph per nonterminal, and can be solved with a fixpoint computation. If the resulting graph is not a tree, then either the constraints are inconsistent, or too few constraints were specified. Such a nesting tree leads to additional edges between the barriers of the NDG. If the PDGs become cyclic due to these additional edges, the constraints on phases were inconsistent. This approach infers a single nesting tree per nonterminal.

The inference of the nesting tree has the advantage that it becomes easier to compose phases. On the other hand, the nesting of phases is typically limited, and has a *purpose*, so it is only a slight burden to specify the nesting fully.

**Overlap prevention.** Sibling phases may not overlap, otherwise it is unclear which evaluation of a node corresponds to which phase. Two phases overlap if either  $inh.x.begin_{\rho_2} \leftarrow^+ inh.x.begin_{\rho_1}$  and  $syn.x.end_{\rho_2} \leftarrow^+ syn.x.end_{\rho_1}$ , or the other way around. However, the dependencies on barriers do not guarantee this property. Given two sibling phases  $\rho_1$  and  $\rho_2$ , we wish to express that either **end**  $\rho_1 \prec$  **begin**  $\rho_2$  or **end**  $\rho_2 \prec$  **begin**  $\rho_1$ . Since siblings are not ordered, we do not know which of the two to take.

Attribute scheduling is actually a reduction of the ordering of phases. If we define phases such that each attribute is declared in a unique phase, then determining the order of phases is attribute scheduling. The order of phases may thus be dependent on context. Therefore, we take a similar approach as Section 4.5 and define the phases graph.

#### 4 AGs with Commuting Rules

**phases graph.** The phases graph of a nonterminal is the phases graph of the master phase of the nonterminal. A phases graph is a DAG where each represents a phase, and an edge from  $a$  to  $b$  means that  $a$  comes before  $b$  in the sequence. A vertex is labelled with the phases graph of its children:

$$g ::= \mathbf{phase} \ \rho \ \bar{g} \ \bar{e} \quad \text{-- phases graph of phase } \rho \text{ (initially the master phase)}$$

$$e ::= g \rightarrow g \quad \text{-- edges connect phases subgraphs}$$

Such a graph  $g$  may have several sources and sinks. However, each path from source to sink must contain precisely all sibling phases, because each path corresponds to an ordered sequence of siblings in the nesting tree of the nonterminal.

Similar as the visits graph, the phases graph contains all consistent ways to interleave phases. An interleaving is consistent when it satisfies the partial order of the phases. In practice, there are only a few phases per nonterminal, with relatively dense constraints, thus the actual required portion of the graph contains little variety. Also, the problem is slightly easier compared to Section 4.5 because each path from source to sink is equally long, and the number of vertices on such a path is known beforehand.

We take a slightly different representation of the phases graph to stress the similarity with the visits graph. The vertices  $s$  of the phases graph represent an ordered sequence of the available  $\mathbf{inh.begin}_\rho$  attributes. An edge is annotated with the attribute that has become available:

$$s ::= \bar{a} \quad \text{-- state of a vertex in the phases graph}$$

$$e ::= s \xrightarrow{a} s \quad \text{-- edge in the phases graph}$$

Note that we can construct the nesting tree from such a sequence.

The purpose of the phases graph is to determine a small number of totally ordered nesting trees for a nonterminal. In a similar way as with visits, we determine a total order on the phases of children of a production given the total order on the phases of the production. For this analysis, we do not add visit-vertices to the PDGs. Instead, we add dependencies between attributes  $\mathit{syn.x.end}_{\rho_1}$  and  $\mathit{inh.x.begin}_{\rho_2}$  for children  $x$  and phases  $\rho_1$  and  $\rho_2$ .

Recall that a vertex in a PDG is available when all its dependencies are available. In this situation, for a begin barrier to be ready, it must have been connected to an end barrier. Figure 4.27 gives the definition. Further, an end-barrier is required when the begin-barriers of the phase and its children are available. Sufficient child-edges must be added so that the (indirect) dependencies of a required end-barrier are available.

**Definition (Selectable).** A begin-barrier is *selectable* when the following conditions are met.

- Its dependencies are available. This implies that the begin-barrier of its parent is available.
- If a begin-barrier of a sibling is available, then so is the end-barrier of that sibling. This ensures that the phases do not overlap.

Options for child-edges can thus be determined by taking the intersection between selectable and required begin-barriers. If this intersection is empty, then the dependencies on

$$\begin{aligned}
\text{avail } \bar{a} \ g \ d &= \text{all } (\text{avail } \bar{a} \ g) (\text{deps } g \ d) \wedge \text{except } \bar{a} \ g \ d \\
\text{except } \bar{a} \ g \langle \text{inh.lhs.y} \rangle &| \neg \text{isBegin inh.lhs.y} \vee \text{inh.y} \in \bar{a} \\
\text{except } \bar{a} \ g \langle \text{inh.x.y} \rangle &| \neg \text{isBegin inh.x.y} \vee (d \in \text{deps } g \langle \text{inh.x.y} \rangle \wedge \text{isEnd } d) \\
\text{except } \bar{a} \ g \langle \text{syn.c.y} \rangle &| \text{True} \\
\text{except } \bar{a} \ g \langle \text{rule } x \rangle &| \text{True} \\
\text{except } \bar{a} \ g \langle \text{child } x \rangle &| \text{True} \\
\text{except } \bar{a} \ g \langle \text{visit } x \ i \rangle &| \text{True} \\
\text{isBegin } \langle k.x.\text{begin}_\rho \rangle & \\
\text{isEnd } \langle k.x.\text{end}_\rho \rangle &
\end{aligned}$$

**Figure 4.27:** The relation *avail* with begin and end barriers.

phases are inconsistent, i.e. forcing the barriers to overlap. If there are multiple options available, we prioritize based on some stable order, such as the order of appearance.

As initial solution, we start with only the begin-barrier of the master phase available for each nonterminal. For the root nonterminals, we construct as initial solution a full path by determining a last-as-possible order given the dependencies of the NDG, and prioritizing based on the stable order mentioned earlier.

**Attribute scheduling.** The set of paths from sink to source is in practice small in the phases graph. The amount of paths is the price to pay for not specifying a total order on phases. For each path in the phases graph, we determine the accompanying nesting tree. The set of these trees form the phase interfaces of a nonterminal. For each child, we specify which phase interface to use, using a type index for the child-rule.

With each phase interface corresponds a specialized version of the NDG and PDGs. We also construct a separate visits graph per phase interface. Due to the additional dependencies imposed by the phases-interface, variability in the visits graph is reduced.

Moreover, on each nesting-tree, we can perform additional attribute scheduling. The attributes are restricted by a partial order, and this order may be strengthened if it does not lead to cycles in the NDG of the nesting tree. For example, we may schedule attributes to the latest possible phase, or avoid scheduling attributes to certain phases. Such heuristics are similar to those discussed for rule scheduling in Section 3.5.2.

**Remarks.** The generalization to phases ensures that our approach is a conservative extension of ordered attribute grammars. A conventional attribute grammar can be expressed by associating all attributes with one phase.

The inference of the phases graph is similar to the inference of the visits graph. It may be possible to combine the inference of both graphs. However, it is not immediately clear how to express heuristics, such as the avoidance of certain phases, in a combined approach.

```

attr Tree  thr unq :: Int           -- declaration of threaded attribute
sem Root
  | Root  root.unq = 1                 -- initial value of threaded attribute
           loc.final = root.unq       -- final value of threaded attribute
sem Tree                                     -- modifications to the threaded attribute
  | Leaf  (loc.myId, lhs.unq) = (lhs.unq, lhs.unq + 1)  lhs.unq  $\diamond$  lhs.unq
  | Bin   l.unq                = lhs.unq                lhs.unq  $\diamond$  l.unq
           r.unq                = l.unq                  l.unq  $\diamond$  r.unq
           lhs.unq              = r.unq                  r.unq  $\diamond$  lhs.unq

```

Figure 4.28: Example of commuting rules.

## 4.9 Commuting Rules

In this section, we present AGs with commuting rules, which are chained rules that can be reordered. These commuting rules provide an abstraction for the use of references in Section 4.3. Given an explicit ordering of rules, the composition of two rules is *commutative* when the two rules are *commutable*, which means that the rules may be swapped in the composition. Rule composition is a conditionally *commutative operator*. Commutable rules represent *commutable operations*. The swapping of rules models side effects, and commutativity facilitates reasoning about the safe use of side effects.

**Syntax.** We assume some conventional AG to start with, and show later how to encode the commutable rules in BarrierAG. Firstly, we introduce *threaded attributes*, which are a special form of chained attributes. Secondly, we introduce commuting rules, which specify a set  $\bar{h}$  of commutable chains the rule participates in:

```

k ::= ...           -- attribute forms (e.g. inh and syn)
  | thr             -- threaded attribute
r ::= ...           -- rules
  | x: p  $\bar{z}_1$  = f  $\bar{z}_2$   $\bar{h}$  -- rule x that commutes over  $\bar{h}$ 
h ::= z1  $\diamond$  z2 -- rule commutes with rules of z1 and z2

```

The syntax  $z_1 \diamond z_2$  specifies that the rule connects chains of  $z_1$  and  $z_2$ . Attribute  $z_1$  must be on an input position, and  $z_2$  on an output position. Also, both must be threaded attributes, and their types must be the same.

The example in Figure 4.28 uses a threaded counter to demonstrate the commuting rules. Each value of *loc.myId* is unique, although it is not guaranteed that a right-sibling of a node has a higher *loc.myId* value. This would be the case if the rules were not commutable.

The example shows as the first rule of *Root* how to provide the initial value of a threaded attribute. This is a rule that defines a threaded attribute, but does not commute over it. Also,



<b>sem</b> $N \mid P$	-- $k$ is a child of production $P$ of nonterminal $N$
$loc.k_x^{ref} :: IORef \tau$	-- local attribute declarations with type
$loc.k_x^{wr}$ <b>barrier</b>	-- local barrier declaration
$loc.k_x^{rd}$ <b>barrier</b>	-- local barrier declaration
$k.x_{ref}^\bullet = loc.k_x^{ref}$	-- copy as reference for child
$k.x_{wr} < loc.k_x^{wr}$	-- restrain read barrier of child
$k_x^{rd} < k.x_{rd}$	-- restrain write barrier of child

**Figure 4.29:** Sketch of the encoding of commuting rules in BarrierAG.

the example shows as the second rule of *Root* how to obtain the final value of a threaded attribute, which is a rule that refers to the value of a threaded attribute, but does not commute over it. Finally, the rules of *Tree* are commutable rules.

As additional requirement, a rule that defines a threaded attribute  $lhs.y$  must mention  $lhs.y$  in its set of commuting chains  $\bar{h}$ . This restriction ensures that the threaded attribute can be represented as an inherited attribute.

The commuting chains  $\bar{h}$  specifies in which chains a rule participates:

$$uses \bar{h} = \{z_1 \mid \langle z_1 \diamond z_2 \rangle \in \bar{h}\}$$

$$defs \bar{h} = \{z_2 \mid \langle z_1 \diamond z_2 \rangle \in \bar{h}\}$$

A rule with  $\bar{h}$  connects  $uses \bar{h}$  to  $defs \bar{h}$ .

**BarrierAG encoding.** We use a BarrierAG encoding as a means to specify the implementation of commuting rules. We represent each threaded attribute  $\mathbf{thr} x :: \tau$  with three attributes in BarrierAG:

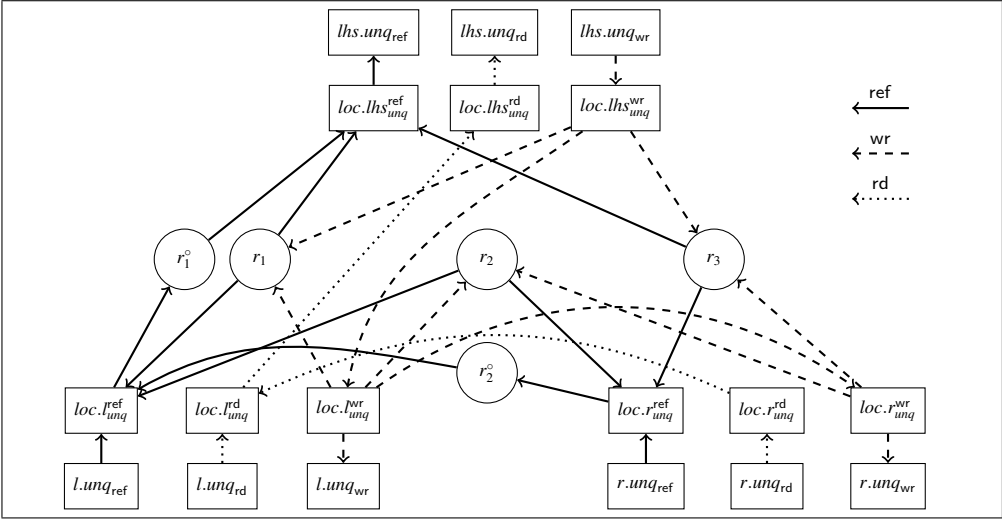
<b>inh</b> $x_{ref} :: IORef \tau$	-- reference to the mutable state
<b>syn</b> $x_{wr}$ <b>barrier</b>	-- all commutable updates before this barrier
<b>inh</b> $x_{rd}$ <b>barrier</b>	-- all non-commutable updates after this barrier

Thus, a threaded attribute is a reference to a mutable state. We ensure in the encoding that rules only depend on the reference, which permits the reordering. As invariant, the write barrier of the child depends on all commuting rules of the child (and its subtree) that update the reference. All non-commuting rules that refer to the reference depend on the read barrier.

For each threaded attribute  $\mathbf{thr} y :: \tau$  of a child  $k$ , we introduce three local attributes, and rules to connect the local attributes with the attributes of the child. Figure 4.29 gives a sketch. Note that  $k_x^{ref}$  is the name of the attribute. The name of the child  $k$  is part of the name of the attribute.

When a rule refers to the threaded attribute of  $k$ , we actually let it refer to the local attributes, as we show below. For notational convenience, we also introduce these local attributes for threaded attributes of  $lhs$ :

#### 4 AGs with Commuting Rules



**Figure 4.30:** Attributes encoding the commutative rules of the production *Bin*.

$$\begin{aligned}
 loc.lhs_x^{ref} &= lhs.x_{ref} && \text{-- copy as reference} \\
 loc.lhs_x^{wr} &< lhs.x_{wr} && \text{-- restrain read barrier} \\
 k.lhs_{rd} &< lhs_x^{rd} && \text{-- restrain write barrier}
 \end{aligned}$$

These rules for *lhs* are the contravariant version of the rules for children.

*Non-commuting write.* When a rule  $x$  with commuting chains  $\bar{h}$  defines a threaded attribute  $thr.k.y$ , but  $thr.k.y \notin defs \bar{h}$ , rule  $x$  serves as *initializer* for the threaded attribute. In the encoding, the defining occurrence  $thr.k.y$  in the left-hand side of rule  $x$  is replaced with  $loc.k.y^{ref \times}$ . Moreover, we add the following dependency rule:

$$loc.k_y^{wr} < loc.k_y^{rd} \quad \text{-- orders writes before reads}$$

Since  $k$  is the start of the chain, and given the invariants on the read and write barriers, this dependency rule ensures that commutable writes take place before non-commutable reads.

*Non-commuting read.* When a rule  $x$  with commuting chains  $\bar{h}$  refers to a threaded attribute  $thr.c.y$ , but  $thr.c.y \notin uses \bar{h}$ , rule  $x$  reads the final value of the threaded attribute. Thus, we replace the occurrence  $thr.c.y$  in the right-hand side of rule  $x$  with  $c_y^{ref \circ}$ , and add the following dependency:

$$loc.c_y^{rd} < \mathbf{rule} \ x \quad \text{-- the read depends on the read barrier}$$

The identifier  $c$  is either a child  $k$  or *lhs*.

*Commuting read and write.* When a rule  $x$  with commuting chains  $\bar{h}$  refers to a threaded attribute  $thr.c.y$ , and  $thr.c.y \in defs \bar{h}$ , we replace the defining occurrence in the left-hand side of  $x$  with  $c_y^{ref \circ}$ . When  $thr.c.y \in uses \bar{h}$ , we replace the occurrence in the righthand side of  $x$  with  $c_y^{ref \circ}$ . Moreover, we insert a rule  $x^\circ$  for each commuting chain, which copies the reference and thus links the chain. Also, we connect the read and write barriers:

$$\begin{array}{ll}
x^\circ : id\ k.y_2^\bullet = id\ c.y_1^\bullet & \text{-- for each } c.y_1 \diamond k.y_2 \in \bar{h} \text{ (copies the reference)} \\
\mathbf{rule}\ x \prec loc.c_y^{wr} & \text{-- for each attribute } y \text{ in } \bar{h} \text{ (rule before write barrier)} \\
loc.c_{y_2}^{wr} \prec loc.k_{y_1}^{wr} & \text{-- for each } c.y_1 \diamond k.y_2 \in \bar{h} \text{ (connect write barriers)} \\
loc.c_{y_1}^{rd} \prec loc.k_{y_2}^{rd} & \text{-- for each } c.y_1 \diamond k.y_2 \in \bar{h} \text{ (connect read barriers)}
\end{array}$$

When  $lhs.y \in defs\ \bar{h}$ , the read and write barrier are not connected, as it would create a cycle. Also, note the contravariant behavior between the read and write barrier.

The barrier attributes and dependency rules enforce the proper ordering of rules that use threaded attributes. Figure 4.30 demonstrates the encoding for the Bin-production. The boxes represent attributes. The circles represent the relevant rules. The barriers and their dependencies are not part of the generated code, and thus do not have a runtime overhead. As a consequence, we actually transformed a chained attribute into an inherited attribute with a reference to a mutable state.

**Referential transparency.** Referential transparency is important for equational reasoning. For AGs, it is also important to ensure that the order of evaluation does not affect the result. In BarrierAG, rules that use updatable attributes break referential transparency. However, with commutable rules, we can establish a weaker version of referential transparency. When two rules commute, the actual values for the attributes that these rules define may be different, but in the context where these rules are defined, the final result, which abstracts from the values of the attributes, may still be equivalent to any ordering of the commutable rules.

The composition of rules, and the context of rules can be made explicit with arrow notation. A commuting rule  $r_1 : (x_1, y_1) = f(x_0, y_0)$ ,  $x_0 \diamond x_1$  and a commuting rule  $r_2 : (x_2, z_1) = g(x_1, z_0)$ ,  $x_1 \diamond x_2$  correspond respectively to the arrows  $(x_1, y_1) \leftarrow f \prec (x_0, y_0)$  and  $(x_2, z_1) \leftarrow g \prec (x_1, z_0)$ . Section 1.3.9 shows that the composition of these rules can be expressed as an arrow:

$$\begin{array}{l}
\mathbf{proc}\ (x_0, y_0, z_0) \rightarrow \mathbf{do} \\
\quad (x_1, y_1) \leftarrow f \prec (x_0, y_0) \\
\quad (x_2, z_1) \leftarrow g \prec (x_1, z_0) \\
\quad \mathbf{returnA}\ (x_2, y_1, z_1)
\end{array}$$

Alternatively, when we reorder  $f$  and  $g$ , and rename the attributes, we obtain the following arrow:

$$\begin{array}{l}
\mathbf{proc}\ (x_0, y_0, z_0) \rightarrow \mathbf{do} \\
\quad (x_1, z_1) \leftarrow g \prec (x_0, z_0) \\
\quad (x_2, y_1) \leftarrow f \prec (x_1, y_0) \\
\quad \mathbf{returnA}\ (x_2, y_1, z_1)
\end{array}$$

This notion can straightforwardly be generalized to rules that commute over many attributes, or define and use many other attributes.

## 4 AGs with Commuting Rules

These two rules are *commutable* over attributes of  $x_0, x_1$  and  $x_1, x_2$  if their compositions are equivalent for a given rule context  $h$ , and  $r_1 \not\prec r_2$ :

$$h \left( \begin{array}{l} \mathbf{proc} (x_0, y_0, z_0) \rightarrow \mathbf{do} \\ (x_1, y_1) \leftarrow f \prec (x_0, y_0) \\ (x_2, z_1) \leftarrow g \prec (x_1, z_0) \\ \mathit{returnA} (x_2, y_1, z_1) \end{array} \right) \equiv h \left( \begin{array}{l} \mathbf{proc} (x_0, y_0, z_0) \rightarrow \mathbf{do} \\ (x_1, z_1) \leftarrow g \prec (x_0, z_0) \\ (x_2, y_1) \leftarrow f \prec (x_1, y_0) \\ \mathit{returnA} (x_2, y_1, z_1) \end{array} \right)$$

If there exists directly or indirectly a dependency between  $r_1$  and  $r_2$  then the rules may not commute. This is for example the case when  $r_1$  defines a (non-threaded) attribute that is used by  $r_2$ , or because of a dependency rule.

The rule context  $h$  is an abstraction of the composition in which the composition of  $f$  and  $g$  is contained. For example,  $h$  can represent the composition of the rules of the entire tree, and thus the rule states that the end result of the computation is not affected. In practice, we take a more abstract notion of  $h$ . For example, the property that all *loc.myId* attributes have a unique value. In Section 2.2 we give some examples of the function  $h$ .

**Remarks.** To reason with commutable rules, we may need to make assumptions about the order of evaluation. This is, for example, the case when the values of the attributes are trace monoids [Diekert and Métivier, 1997]. Phases can be used for this purpose.

The identification of commuting rules may be relevant for the parallel and incremental evaluation of attribute grammars. Chained attributes sequentialize code, whereas commuting rules allow more interleaving. Similarly, during incremental evaluation, changes in a subtree that appears earlier in the evaluation may be lifted over a later subtree if these changes are visible in a threaded attribute.

Commutable rules can also be used to collect statistics or other runtime properties about the evaluation process, such that the attribute-dependencies of the collecting rules have only a minor influence on the evaluation process. For example, a count of the nodes of the tree traversed so far may be an indication of how much work has been done.

In case of type inference, substitutions may be represented as a threaded attribute, so that the threading of the substitution does not influence the order of evaluation. Traditionally, unification is only a commutable operation when all unifications succeed. By encoding the substitution as a graph structure, it is possible to make unification a commutable operation also in the case of a type conflict [Heeren, 2005].

## 4.10 Related Work

The ability to compose attribute grammars is an important benefit that attribute grammars offer. In fact, AGs are so easily composed that the compositions may accidentally become inconsistent, i.e. have attributes with a cyclic definition. Knuth [1968] proves that an AG is well-defined if and only if the dependency graphs of productions are cycle-free according to his refined algorithm (Knuth-2). When the dependency graphs are cycle-free according to Knuth's original algorithm (Knuth-1) then the AG is well-defined, but not necessarily

the other way around. These are static properties of AGs that provide guarantees that the evaluation of an AG terminates.

Knuth-2 uses a dependency graph per production/child production combination, in contrast to a single dependency graph per production as the Knuth-1 approach uses. Knuth-2 leads to an approximate number of dependency graphs per production in the order ( $p^b$ ), where  $p$  is the number of productions of the child nonterminals, and  $b$  is the number of children. In practice, e.g. for the let-production of a lambda calculus,  $p$  is rather large ( $p > 10$ ), but  $b$  is typically small ( $b \leq 2$ ). However, we usually define a fine granularity of nonterminals so that distinct dependency graphs per production/child production combination does not offer an advantage over a single graph per production.

In our experience, AGs are either necessarily cyclic, or are cycle-free in both Knuth-1 and Knuth-2. If the AG is not cycle-free with Knuth-1, then this is an indication that the AST does not have sufficient structure, which is not likely in strongly typed languages. In the first case, on-demand evaluation may still yield results for attributes, although it is the responsibility of the programmer to ensure this. In the second case, we know that an evaluation order exists, and can use a statically ordered evaluation algorithm. A statically ordered evaluation algorithm is likely to exhibit better time and space behavior, and actually permits minor assumptions about the evaluation order to be made.

Kastens [1980] presented an approach to infer a visit interface per nonterminal. Unfortunately, when using the Kastens approach, we often encounter cycles that are induced by the scheduling as resulting from Kastens' scheduling algorithm. These induced cycles hamper compositionality, because as remedy, we need to add artificial dependencies between attributes to the AG to control the scheduling. Also, the effect of scheduling is not visible in the original rules of the grammar, which makes such cycles very hard to understand and resolve.

The approach by Kennedy and Warren [1976] can find a solution, but may possibly result in an exponentially large solution. In practice the solution is not so large: In our experience, this approach works very well for small AGs. For large AGs, however, the exponential behavior may show up. To counter this behavior, we provide sufficient mechanisms to restrict the set of solutions.

## 4.11 Conclusion

This chapter demonstrates one of the great strengths of attribute grammars: the ability to statically analyse the grammar with abstract interpretations. As one of the main contributions of this chapter, we reformulated the approach of Kennedy and Warren [1976] so that it can be used to generate code for a strongly typed, purely functional host language. For absolutely non-circular AGs, this approach finds a way to order the rules statically.

By using such an approach, it is not needed to explicitly schedule attributes and rules, unless these need to be given special properties. We introduce the notion of phases for this. Since the ordering of attributes and rules can be inferred, we may omit such details from our specification. Consequently, the specification becomes more concise and easier to compose, which is beneficial from an engineering point of view.

#### 4 AGs with Commuting Rules

The price that we pay is that such an analysis can only give an answer per context, such as a context dependent phase or visit interface, and a nonterminal may be in exponentially many contexts. A programmer typically has some evaluation order in mind, thus is usually able to specify a single order that works in all contexts, which may reduce the size of the generated code, compile time, and the execution time. On the other hand, if there is not a single order that works in all contexts, then a programmer is not likely to be able to keep track of all the possibilities manually. We presented phases as a means to specify knowledge about the order of evaluation, without going into the fine details, as in Chapter 3.

Also, for some problems, the order imposed by attribute dependencies may be too restrictive. We presented commuting rules as a means to loosen some restrictions. Commuting rules can be used when a number of rules form a chain, and the individual ordering of the rules is not relevant for the result. A typical example is a sequence of rules that provides unique numbers to nodes in the tree. When the requirement is only that each number is unique, the actual order in which numbers are handed out is does not invalidate that requirement.

As future work, more quantitative insight is needed about the effects of dependency analysis on the performance of the generated code. In earlier experiments, the impact seemed negligible, although the results differed from one AG to another. Also, we need more insight which heuristics have impact on the size of the visits graph. For example, it appeared that our approach in which we only compute what is needed resulted in less paths in the visits graph than the approach where we compute as much as possible. However, more quantitative evidence is required to draw such conclusions.

Another direction of future work is an extension of the Kennedy-Warren approach with support for cyclic AGs. Given a collection of cyclic PDGs, we can determine which attributes of the NDGs are mutually dependent. By grouping these attributes in a separate phase that uses lazy evaluation (for example) as evaluation algorithm, we can support a combination of a cyclic and non-cyclic AGs.

## 5 Derivation Tree Construction

Type inference on a program is the gradual process of constructing a *typing derivation*, which is a proof that relates the program to a type. During this process, inference algorithms analyze the intermediate states of the typing derivation to direct the construction of the proof.

Since a typing derivation is a decorated tree, we aim to use attribute grammars to implement type inference. In their present form, it is hard to express type inference in attribute grammars, because attributes are defined in terms of the final state of the decorated tree.

We present the language RulerCore, a conservative extension to ordered, higher-order attribute grammars, that permits both the structure and attributes of the tree to be defined based on intermediate states of the tree. We show that both iteration-based and constraint-based inference algorithms can be expressed straightforwardly in RulerCore.

### 5.1 Introduction

Attribute grammars (AGs) are traditionally used to specify the static semantics of programming languages [Knuth, 1968]. Moreover, when semantic rules of an AG are written in a general purpose programming language, the AG can be compiled into an (efficient) multi-visit tree walk algorithm that implements the specification [Kennedy and Warren, 1976, Kastens, 1980].

We implemented a substantial part of the Utrecht Haskell Compiler (UHC) [Dijkstra and Swierstra, 2004, Fokker and Swierstra, 2009] with attribute grammars using the UUAG system [Löh et al., 1998]. Haskell [Hudak et al., 1992] is a purely functional programming language, with an elaborate and expressive type system. We also compile our attribute grammars to Haskell. The ideas presented in this chapter, however, are not restricted to any particular language.

Attribute grammars benefit the implementation of a compiler for several reasons. Firstly, the evaluation order of semantic rules is determined automatically, unrelated to the order of appearance. Rules may be written separately from each other, and grouped by aspect, which makes attribute grammars highly composable [Viera et al., 2009, Saraiva, 2002]. Secondly, semantic rules for idiomatic tree traversals (such as: topdown, bottom-up, and in-order) can be inferred automatically, thus allowing for concise specifications. These advantages play an important role in the UHC project [Dijkstra et al., 2009].

Two essential components of UHC's type inferencer, polymorphic unification and context reduction, would benefit from an AG-based implementation. For example, when polymorphic unification is defined as an AG, many of its required attributes can be automatically provided by the AGs of expressions and declarations. However, we implemented these components directly in Haskell, because it is not obvious how to express these as an attribute grammar.

## 5 Derivation Tree Construction

These two components present two challenges to attribute grammars. Firstly, the grammar needs to produce typing derivations. The structure of such a derivation depends on what is known about types, and this information gradually becomes available during inference, typically as a result of unifications. This requires a mixture of tree construction and attribute evaluation, which are normally separate tasks if one takes an AG view. Secondly, the construction of a proof of a subgoal may need to be postponed when it depends on a type that is not known yet. After more of the structure of the proof is determined, the type may become known and the postponed construction can continue.

An evaluator for an attribute grammar starts from a given tree (usually constructed by the parser), and evaluates the attributes using a fixed algorithm. We present AG extensions to customize the algorithms, without losing the advantages that AGs offer. More precisely, our contributions are:

- We present the language RulerCore, a conservative extension of ordered attribute grammars. It has three concepts to deal with the above challenges:
  - We exploit the notion of visits to the tree. In each visit, some attributes are computed, as we explained in Chapter 3. Visits can be done iteratively. The number of iterations can be specified based on the values of attributes.
  - We define abstract grammars on the structure of typing derivations. Productions are chosen based on the values of attributes. Moreover, we present *clauses*, which allows the choice of a production to be refined per visit.
  - Derivation trees are first class values in RulerCore. They can be passed around as attributes, and can be inspected by visiting them.

In Section 5.3, we define a denotational semantics for RulerCore via a translation to Haskell.

- In Chapter 3, we gave an introduction to RulerCore. In this chapter, we show how RulerCore can be used to express type inference. Section 5.2 presents an extensive example that motivates the design of RulerCore.
- An implementation of RulerCore is available at: <https://svn.science.uu.nl/repos/project.ruler.papers/archive/ruler-core-1.0.tar.gz>. It includes several examples. The implementation supports both statically ordered and demand driven evaluation of attributes.
- We compare our approach with other attribute grammar approaches (Section 5.4) as a further motivation for the need for RulerCore’s extensions.

## 5.2 Motivation

In this section, we show how to implement a small compiler for an example language we named SHADOW, written with attribute grammars using RulerCore. The implementation of SHADOW poses exactly those challenges mentioned in the previous section, while being small enough to fit in this chapter.



### 5.2.1 Example: the Shadow-language

We take for SHADOW the simply-typed lambda calculus, with two small modifications: we annotate bindings with a unique label  $u$  (i.e.  $\lambda x^u.$ ), and allow identifiers to refer to *shadowed* bindings. For example, in the term  $\lambda x^{u_1}.\lambda x^{u_2}.f x$ , the expression  $x$  normally refers to the binding annotated with  $u_2$ . However, if the expression cannot be typed with that binding, we allow  $x$  to refer to the shadowed binding annotated with  $u_1$  instead, if this interpretation would be well-typed<sup>1</sup>. The interpretation of this expression is thus by default  $\lambda u_1.\lambda u_2.f u_2$ , but under some conditions (defined more precisely further below) it may be  $\lambda u_1.\lambda u_2.f u_1$ .

The compiler for SHADOW takes an expression (of type  $ExprS$ ), type checks it with respect to the environment  $Env$ , and maps it to an expression (of type  $ExprT$ ) in the simply-typed lambda calculus. It is realized as a function  $compile :: Env \rightarrow ExprS \rightarrow ExprT$ :

```
-- concrete syntax and its abstract syntax in Haskell
eS ::= x | eS eS |      data ExprS = VarS Ident | AppS ExprS ExprS
      λxu.eS                | LamS Ident Ident ExprS
eT ::= u | eT eT |      data ExprT = VarT Ident | AppT ExprT ExprT
      λu.eT                | LamT Ident Ident ExprT
τ ::= α | Int | τ → τ    data Ty = TyVar Var | TyInt | TyArr Ty Ty
```

Before delving into the actual implementation, we first give a specification of the type system, together with translation rules.

$$\boxed{\Gamma \vdash e_S : \tau \rightsquigarrow e_T}$$

$$\frac{\text{innermost } x^u \text{ of all:} \quad x^u : \tau \in \Gamma}{\Gamma \vdash x : \tau \rightsquigarrow u} \text{VAR} \quad \frac{\Gamma \vdash f : \tau_1 \rightarrow \tau_2 \rightsquigarrow f'}{\Gamma \vdash f a : \tau_2 \rightsquigarrow f' a'} \text{APP} \quad \frac{\Gamma, x^u : \tau_1 \vdash e : \tau_2 \rightsquigarrow e'}{\Gamma \vdash \lambda x^u.e : \tau_1 \rightarrow \tau_2 \rightsquigarrow \lambda u.e'} \text{LAM}$$

Each lambda is assumed to be annotated with a unique identifier  $u$ . Rule VAR is rather informal<sup>2</sup>. Of all the bindings for  $x$  with the right type  $\tau$ , the innermost one is to be chosen. Its annotation  $u$  is used as name in the translation. The rule APP is standard. In rule LAM, the type of a binding is appended to the environment. The annotation of the binding is used as the name of the binding in the translation.

Given an (empty) environment, a SHADOW expression, and optionally a type, we can manually construct a derivation tree using these translation rules. The lookup of a binding poses a challenge due to context sensitivity. For example, for  $\lambda x^{u_1}.\lambda x^{u_2}.f x$ , the choice between translations  $\lambda u_1.\lambda u_2.f u_1$  and  $\lambda u_1.\lambda u_2.f u_2$  depends on what the program, where this expression occurs in, states about the type of  $f$  and the type of the entire expression by itself. When

<sup>1</sup> The language SHADOW can be used to model typed disambiguation of duplicately imported identifiers from modules. However, SHADOW is only an example. Its design rationale is out of the scope of this section.

<sup>2</sup> Actually, the specification itself is incomplete and informal. We stress that our goal is not to rigorously discuss and prove properties about SHADOW. Instead, we show RulerCore and its concepts. The translation for SHADOW acts as illustration.

## 5 Derivation Tree Construction

the context imposes insufficient restrictions to find a unique solution, the `VAR` states that we should default to the innermost possibility<sup>3</sup>.

### 5.2.2 Relation to Attribute Grammars

We focus on writing an implementation for the above translation rules with attribute grammars using `RulerCore`. For each relation in the above specification, we introduce a nonterminal in the `RulerCore` code. The parameters of the relations become attributes of these nonterminals, thus also the expression part which is normally implicit in an AG based description. Derivation rules become productions, and their contents we map to semantic rules. The productions do not contain terminals: only the values of attributes determine the structure of the derivation tree. Thus, the grammar defines the language of derivation trees for the translation rules. Note that this differs from the standard AG approach, where a single specific parameter of the relations fully determines the shape of the derivation tree.

Operationally, the algorithm, which specifies the construction of the derivation tree, picks a production, recursively builds the derivations for the children of the production, computes attributes, and if there is a mismatch between the value of an attribute and what is expected of it, backtracks to the next production.

We treat productions a bit differently in order to capture the gradual process of type inference. Final decisions about what productions are chosen to make up the derivation tree cannot be made until sufficient information is available. Therefore, we construct the derivation tree in one or more sequential passes called visits.

As key feature of `RulerCore`, the grammar may contain productions per visit of a nonterminal. To make the distinction clearer, we call these productions *clauses*. During the construction of the part of a derivation for a visit, we try to apply the available clauses to build the portion of the derivation tree for that visit. When successful, we finalize the choice for that clause (similar to the cut in Prolog). The next visit can thus assume that those parts of the derivation tree constructed in previous visits is final. Moreover, we often wish to repeat a visit when type information was discovered that sheds new light upon decisions taken earlier during the visit. The upcoming example code shows this behavior several times.

In a conventional AG, each node in the AST is associated with exactly one production. In `RulerCore`, however, we may at each visit *refine* the production that is associated with the node. So, we can regard our approach as having our productions organized as the leafs of a tree of clauses, and at each next visit we specialize the choice by going down one of the paths in the tree. Paths in this tree determine actual productions.

A `RulerCore` program is a Haskell program augmented with attribute grammar code. We generate plain Haskell code from such a program. For each production of `RulerCore` we generate a coroutine in Haskell. These coroutines are encoded as continuations. The following is a code snippet of a `RulerCore` program. We explain the syntax further below:

```
module MyCompiler where  
data ExprS = ... -- Haskell data declaration
```

---

<sup>3</sup> This example has a strong connection to context reduction in Haskell. The inference rules are type-directed. Such rules may be overlapping, and the choice of which rule to apply in a typing derivation may be ambiguous.

```

itf TransExpr ... -- RulerCore nonterminal and attribute declaration
itf TransExpr ... -- additional attribute declarations for nonterminal
ones = 1 : ones -- Haskell binding
translate = sem transExpr :: TransExpr -- embedded production
                ... rules ... -- rules of RulerCore
sem transExpr ... rules ... -- additional rules for production
sem transExpr ... rules ... -- even more rules

```

With an *itf*-block, we define a nonterminal and its attributes. With a *sem*-block, we we define a production, and its clauses and rules, inside a Haskell expression. The *sem*-block is substituted with the coroutine that is generated for *transExpr*. This coroutine is thus a Haskell expression that, and is bound to the Haskell identifier *translate*. Additional clauses and rules can be given in separate toplevel *sem*-blocks.

The coroutines that we generate from productions are known as visit functions [Swierstra and Alcocer, 1998]. Inputs to and outputs of the coroutine represent inherited and synthesized attributes respectively. Clauses are mapped to function alternatives of the coroutine. The internal state of the coroutine represents the derivation tree. This state contains instances of coroutines (their closures) that represent the children. An invocation of such a visit function corresponds to a visit in the attribute grammar description. An invocation of a visit function of a root nonterminal thus corresponds to the partial construction of the root node of the derivation tree. Section 5.3 shows the translation to Haskell.

In RulerCore, a production is the root of a tree of clauses. Thus, we can represent a production of a conventional grammar as a production in RulerCore. Alternative, as we do in the example, we can also introduce only one production per nonterminal, and use clauses to represent productions of a conventional grammar. The distinction is mainly technical: productions can be used when the tree is determined before attribute evaluation (the productions form an algebra), whereas clauses can be used to determine the structure of the tree based on attribute values.

### 5.2.3 Typing Expressions

Figure 5.1 gives a rudimentary sketch of a derivation tree and some of the nonterminals and productions that we introduce. The root node corresponds to production *wrapper* of nonterminal *Compile*. It has a child which is related to production *transExpr* of nonterminal *TransExpr*. Production *transExpr* consists of clauses *exprVar*, *exprLam*, etc. for the various forms of syntax of SHADOW. In locating an identifier in the environment, three nonterminals play a role. Clause *lookupTy* of nonterminal *Lookup* has a list of children, each with nonterminal *LookupMany*. Each child is associated with a clause *lookupLam* of nonterminal *LookupOne*, and represent a choice for a binding. The dotted line points to that binding. At the end of inference, at most one of these choices remains per child with nonterminal *Lookup*.

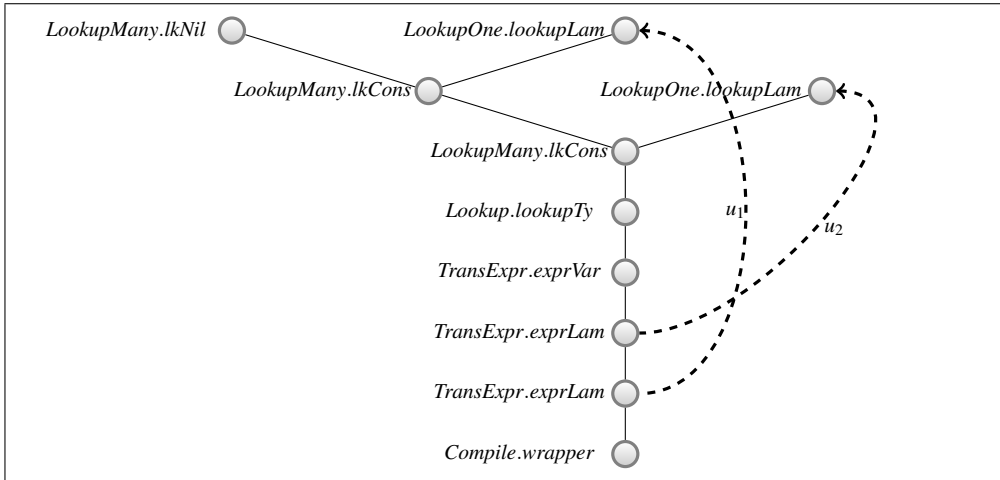
We declare a type *TransExpr* for the production *transExpr*, which describes the *interface* of a nonterminal. The interface declares the visits and attributes of a nonterminal:

```

itf TransExpr          -- declaration of nonterminal TransExpr
inh env :: Env        -- inherited attr (belongs to some visit)

```

## 5 Derivation Tree Construction



**Figure 5.1:** Sketch of a derivation tree for  $\lambda x^{u_1}. \lambda x^{u_2}. x$

```

visit dispatch      -- declares a visit
  inh expr :: ExprS -- inherited attr (belongs to visit dispatch)
type Env = Map Ident [LookupOne] -- shown further below

```

In this case, a production of a nonterminal *TransExpr* has a single visit named *dispatch*, and two inherited attributes. The attribute *expr* contains the expression to translate. We define more attributes and visits on *TransExpr* below.

The visits are totally ordered based on value-dependencies between attributes that are derived from the clauses in the whole program. This order is constructable when the attribute dependencies are acyclic. Chapter 3 explains how to derive this order, and Chapter 4 generalizes that approach. Attribute *expr* is declared explicitly for visit *dispatch* (note the indenting). The *env* attribute is not declared for a particular visit. The latest visit it can be allocated to is determined automatically.

From the interface of a nonterminal, a Haskell type for the coroutines is generated. Also, wrapper functions (Section 1.3.1) to invoke the coroutines and access or provide values for attributes from the Haskell code are generated from the interface.

We introduce a production *transExpr* with nonterminal *TransExpr* using a sem-block, embedded in Haskell code. This sem-block is translated to a coroutine in plain Haskell:

```

-- embedded toplevel Haskell code:
translate = sem transExpr :: TransExpr

```

We bind it to the Haskell name *translate*, such that we can refer to it from the Haskell code. The nonterminal name *transExpr* is global to the entire program. The Haskell name *translate* follows Haskell's scoping rules.

If we visit a *transExpr*-node for the first time, we have to see what kind of expression we have at hand. We do so by defining a number of alternative ways to deal with the node by

introducing a couple of named clauses. For each of the clauses, we subsequently introduce further sem-rules to determine what has to be done. Unlike most of the other code that we give in this section, the order of appearance is relevant for clauses. Operationally, clauses are tried in that order:

```

sem transExpr                -- production with clauses
  clause exprVar of dispatch -- typical name of first visit
  clause exprApp of dispatch
  clause exprLam of dispatch
sem exprVar -- clause with rules (or clause of next visit)
  -- semantic rule (i.e. to match against attributes)
  -- semantic rule (i.e. to define a child)
  -- perhaps clause of next visit (in scope of this clause)

```

Clauses provide a means of scoping. For example, we typically declare clauses of the next visit in the scope of the parent clause (i.e. clause taken at the previous visit). These inherit all the attributes and children in scope of that clause. Otherwise, they only inherit the common attributes and children. This enforces as well that the embedded clause takes place after the enclosing clause.

With a clause, we associate a couple of semantic rules, all of which may fail and cause backtracking, may have an effect on the derivation tree we are constructing, or lead to a runtime error.

- **match** *pattern = code* -- match-rule
- **match** (*VarS loc.nm*) = *lhs.expr* -- example

The Haskell pattern *pattern* must match the value of the right hand side. Evaluation of the rule requires the full pattern match to take place, or causes a backtrack to the next clause.

Variables in the pattern refer to attributes, and have the form *childname.attrname*. The child name *lhs* is reserved to refer to the attributes of the current node. Furthermore, child name *loc* is a virtual child that conveniently stores local attributes, analogously to local variables. In the example, *lhs.expr* thus refers to the inherited attribute *expr* of the current node.

- *pattern = code* -- assert-rule (not prefixed with a keyword)

The meaning of an assert-rule is similar to the match-rule, except that the match is expected to succeed. If not, its evaluation aborts with a runtime error.

- **child** *name :: I = code* -- child-rule
- **child** *fun :: TransExpr = translate* -- example

In contrast to a conventional attribute grammar, we construct the tree during attribute evaluation. The rule above creates a child with the given *name*, described by a non-terminal with interface *I*, and defined by the coroutine *code*. For example, *code* could

## 5 Derivation Tree Construction

be the expression *translate*, or a more complex expression. Further below, we show an example where the code for a child is provided by an attribute. Evaluation of the child rule creates a fresh instance of this coroutine. This child will thus have its own set of attributes defined by *I*.

- **default** *name* [= *f*] -- default rule

Provides a default definition for all synthesized attributes named *name* of the production and all inherited attributes of the children that are in scope. This default definition applies only to an attribute if no explicit definition is given. We come back to this rule further below.

We introduce more forms of rules further below.

The evaluation order of rules is determined automatically based on their dependencies on attributes. Rules may refer to attributes defined by previous rules, including rules of clauses of previous visits. Similarly, attributes are mapped automatically to visits based on requirements by rules. Cyclic dependencies are considered to be a static error. The rules may be scheduled to a later visit, except for match-rules. These are scheduled in the visit of the clause they appear in. Visits to children are determined automatically based on dependencies of attributes of the children. If a visit to a child fails, which is the case when none of the children's clauses applies, the complete clause backtracks as well.

The following sem-block defines rules for clause *exprVar*. It states that the value of attribute *lhs.expr* must match the *VarS* constructor:

```
sem exprVar -- clause exprVar of nonterminal TransExpr
  match (VarS loc.nm) = lhs.expr -- if succesful, defines loc.nm
```

An attribute grammar distinguishes two categories of attributes: inherited and synthesized. The names of attributes within the same category need to be distinct. Attribute variables in patterns refer to *synthesized* attributes of *lhs*, or *inherited* attributes of the children. Likewise, attribute variables in the right-hand side of a match refer to *inherited* attributes of *lhs* or *synthesized* attributes of the children. This ensures that attribute occurrences are uniquely identifiable.

The following clause for *exprApp* demonstrates the use of child-rules. It introduces two children *f* and *a* with interface *TransExpr*, represented as instances of the *translate* coroutine:

```
sem exprApp -- clause exprApp of nonterminal TransExpr
  match (AppS loc.f loc.a) = lhs.expr -- test for AppS
  child f :: TransExpr = translate -- recurses on f
  child a :: TransExpr = translate -- recurses on a
  f.expr = loc.f -- passes loc.f as expr-attribute
  a.expr = loc.a -- passes loc.a as expr-attribute
  f.env = lhs.env -- passes the environment topdown
  a.env = lhs.env -- passes the environment topdown
```

The last two lines express that the environment is passed down unchanged. We may omit these rules, and write the rule **default** *env* instead. When a child has an inherited attribute *env*, but no explicit rule has been given, and the production has *lhs.env*, then that value is automatically passed on:

```
sem transExpr -- for all clauses of transExpr
default env
```

There are several variants of default-rules. We show further below a default rule for synthesized attributes.

We skip the clause *exprLam* for now, and consider types and type inference first. As usual with type inference, we introduce type variables for yet unknown types, and compute a substitution that binds types to these variables:

```
itf TransExpr -- extends the interface of TransExpr
syn ty :: Ty -- synthesized attr in unspecified visit
chn subst1 :: Subst -- chained attr in unspecified visit
data Subst -- left implicit: a mapping from variables to types
```

The chained attribute *subst<sub>1</sub>* stands for both an inherited and synthesized attribute with the same name. We can see this as a substitution that goes in, and comes out again updated with new type information that became available during the visit. We get automatic threading of the attribute through all children that have a chained attribute with this name, using the default-rule:

```
sem transExpr default subst1
```

To deal with types and substitutions, we define several helper nonterminals:

```
itf Lookup -- finds a pair  $(nm, ty') \in env$ 
inh nm :: Ident -- such that  $ty'$  matches  $ty$ .
inh ty :: Ty
inh env :: Env

itf Unify -- computes a substitution  $s$  such
visit dispatch -- that  $s(ty_1)$  equals  $s(ty_2)$ , if
inh ty1 :: Ty -- possible. Attributes for the
inh ty2 :: Ty -- substitution and errors added below.

itf Fresh -- produces a fresh type
syn ty :: Ty
chn subst :: Subst

lookup = sem lookupTy :: Lookup
unify = sem unifyTy :: Unify
fresh = sem freshTy :: Fresh
```

The implementation of *fresh* delegates to a library function *varFresh* on substitutions:

## 5 Derivation Tree Construction

```
sem freshTy
  (lhs.subst, loc.var) = varFresh lhs.subst
  lhs.ty = TyVar loc.var
```

We did not explicitly declare any visits for nonterminal *Fresh*. Therefore, it consists of a single anonymous visit. When a production does not specify a visit-block, an anonymous visit-block is implicitly defined. Similarly, when a production does not define clauses for a production, an anonymous clause-block is implicitly defined.

We can wrap an application of this function, including a data constructors, as a production, and represent an application of this function as child of the production (Section 1.3.7). This is convenient in case of *fresh*, because we use default rules to automatically deal with the substitution attribute.

Both *fresh* and *lookup* are of use to refine the implementation of *exprVar*. With *fresh* we get a fresh variable to use as the type of the expression. The *Lookup*-child then ensures that at some point this fresh type is constrained in the substitution to the type of a binding for the variable:

```
sem exprVar  -- repeated sem-block: extends previous one
  child fr :: Fresh = fresh
  child lk :: Lookup = lookup
  lk.nm = loc.nm  -- pass loc.nm to lk (loc.nm matched earlier)
  lk.ty = fr.ty   -- pass the fresh type to lk
  lhs.ty = fr.ty  -- also pass it up
```

We pass the substitution to child *fr*, and pass the resulting substitution upwards to the parent node:

```
sem exprVar
  sem exprVar  rename subst := subst1 of fr
  fr.subst = lhs.subst1  -- pass down
  lhs.subst1 = fr.subst  -- pass up
```

Recall that *subst<sub>1</sub>* is a chained attribute, hence there is an inherited *lhs.subst<sub>1</sub>*, and a synthesized *lhs.subst<sub>1</sub>*. These names are not ambiguous: the right hand side of the rule refers to the inherited attribute, the left hand side to the synthesized. With a *rename*-rule, we rename attributes of children to choose a more convenient name, for example to benefit from default-rules. The two explicit rules may actually be omitted, because of default-rule mentioned earlier.

In the application clause, we use the *Unify* nonterminal to express that various types should match:

```
sem exprApp
  child fr :: Fresh = fresh
  child u  :: Unify = unify
  u.ty1 = f.ty
```



```

u.ty2 = TyArr a.ty fr.ty
lhs.ty = fr.ty
rename subst := subst1 of fr -- for default-rule

```

Rules for the substitution may be omitted. The default-rule threads it properly through the *fr* and *u* children, which (after renaming) both have a *subst<sub>1</sub>* chained attribute.

## 5.2.4 Unification

So far, the example can be implemented with most attribute grammar systems that operate on a fixed abstract syntax tree [Dijkstra and Swierstra, 2004, 2006b]. In the above example, the choice of productions solely depends on the *expr* inherited attribute. The attribute grammar is directly based on the grammar of expressions. In the remainder of this section, we move beyond such systems. For unification, we allow a selection of clauses based on two inherited attributes: the attributes *ty<sub>1</sub>* and *ty<sub>2</sub>* of nonterminal *Unify* defined above.

The idea behind unification is to recursively compare these types. If one is a variable, then the other type is bound to that variable in the substitution:

```

sem unifyTy
  clause matchEqVars of dispatch -- the same variables
  clause matchVarL   of dispatch -- variable on the left
  clause matchVarR   of dispatch -- variable on the right
  clause matchArr     of dispatch -- both an arrow
  clause matchBase    of dispatch -- both the same constant
  clause matchFail    of dispatch -- failure

```

To implement these clauses, we need additional infrastructure to obtain the free variables of a type, and bind a type in the substitution. We omit here the actual implementation, since the implementation similar to other examples in this section:

```

itf Ftv inh ty    :: Ty    -- determines free vars of ty
      inh subst    :: Subst  -- after applying the subst
      syn vars     :: [Var]
itf Bind inh var  :: Var    -- appends to subst:
      inh ty       :: Ty     -- [var := ty]
      chn subst    :: Subst
ftv = sem ftv :: Ftv -- implemented with RulerCore
bind = sem bind :: Bind -- wrapper around library fun

```

We define several additional attributes on the *Unify* nonterminal. For the synthesized attributes *success* and *errs*, we give a default definition of the form **default** *attr* = *f*. This function *f* gets as its first parameter a list of values of attributes *attr* of the children that have this attribute. If *f* is not given, we use the Haskell function *last* for *f* (Section 2.1):

```

itf Unify
  visit dispatch

```

## 5 Derivation Tree Construction

```

chn subst1 :: Subst
syn success :: Bool -- True iff unification succeeds
syn changes :: Bool -- True iff any variables were bound
visit outcome
inh subst2 :: Subst -- take subst2 more recent as
syn errs    :: Errs  -- subst1 for better error messages

sem unifyTy
default success = and -- and [] = True
default changes = or  -- or [] = False
default errors  = concat

loc.ty1 = tyExpand lhs.subst1 lhs.ty1 -- apply subst one level
loc.ty2 = tyExpand lhs.subst1 lhs.ty2 -- apply subst one level

```

The inherited types need to be compared with what is known in the substitution to ensure that we do not bind to a variable twice. Hence, we introduce attributes *loc.ty<sub>1</sub>* and *loc.ty<sub>2</sub>*, which are computed by applying the substitution to *lhs.ty<sub>1</sub>* and *lhs.ty<sub>2</sub>*. Their values are shared among all clauses and are computed only once. We match on these values to select a clause:

```

sem matchEqVars -- applies if we get two equal vars
  match True = same lhs.ty1 lhs.ty2 ∨ same loc.ty1 loc.ty2
  -- embedded Haskell code:
  same (TyVar v1) (TyVar v2) | v1 ≡ v2 = True
  same _ _ = False
sem matchVarL -- a yet unknown type left
  match (TyVar loc.var) = loc.ty1
  loc.ty = loc.ty2
sem matchVarR -- a yet unknown type right
  match (TyVar loc.var) = loc.ty2
  loc.ty = loc.ty1
sem matchVarL matchVarR -- common part of above
  child fr :: Ftv = ftv -- determine free fr.vars
  fr.ty = loc.ty -- of loc.ty
  child b :: Bind = bind -- add substitution
  b.var = loc.var -- [loc.var := loc.ty]
  b.ty = loc.ty
  rename subst := subst1 of fr b
  loc.occurs = loc.var ∈ fr.vars -- occur check
  lhs.subst1 = if loc.occurs then lhs.subst1 else b.subst1
  lhs.success = ¬ loc.occurs
  lhs.changes = ¬ loc.occurs
  lhs.errs = if loc.occurs
             then [CyclErr lhs.subst2 loc.var loc.ty]

```

```

else []
sem matchArr --  $t_1 \rightarrow t_2$  left and  $t_3 \rightarrow t_4$  right
  match (TyArr  $t_1$   $t_2$ ) = loc.ty1
  match (TyArr  $t_3$   $t_4$ ) = loc.ty2
  child  $l :: \text{Unify}$  = unify -- recurse with argument types
  child  $r :: \text{Unify}$  = unify -- recurse with result types
   $l.ty_1 = t_1 ; l.ty_2 = t_3 ; r.ty_1 = t_2 ; r.ty_2 = t_4$ 
sem matchBase -- applies when e.g. both are TyInt
  match True = loc.ty1  $\equiv$  loc.ty2
sem matchFail -- mismatch between types
  lhs.success = False
  lhs.errs = [UnifyErr lhs.subst2 lhs.ty1 lhs.ty2]

```

The clauses of *unifyTy* are total, thus there is always one that applies, with *matchFail* as fallback. The visits to unification thus always succeed. Potential problems that arose during unification can be inspected through attributes *success* and *errs*.

We now have the mechanisms ready to deal with the case of a lambda expression. For the type of the binding, we introduce a fresh type *fr.ty*, and add this type together with the name to the environment:

```

sem exprLam
  match (LamS loc.nm loc.u loc.b) = lhs.expr
  child  $b :: \text{TransExpr}$  = translate -- recurse
  child  $fr :: \text{Fresh}$  = fresh
  rename subst := subst1 of fr
   $b.expr = loc.b$ 
   $b.env = \text{insertWith } (++) \text{ loc.nm } [loc.lk] env$  -- append
   $lhs.ty = \text{TyArr } fr.ty \ b.ty$  -- result type is  $fr.ty \rightarrow b.ty$ 
   $loc.lk = \text{sem lookupLam} :: \text{LookupOne}$  -- see below

```

Environments are treated in an unconventional way. Instead of transporting the information needed to construct the lookup-derivation tree in *exprVar*, the environment transports a coroutine *loc.lk* defined in *exprLam*. We define the production *lookupLam* locally in *exprLam*, such that the rules of *lookupLam* have access to the attributes of *exprLam*:

```

itf LookupOne -- nonterminal of nested production lookupLam
  visit dispatch inh ty :: Ty

```

The idea is that we instantiate this coroutine at the *exprVar*, then pass it the expected type of the expression, and determine if the expected type matches the inferred type of the binding. The rules for this nested production (shown further below) have access to the local state (i.e. attributes) of the enclosing production. At the binding-site, we have information such as the type and annotation of the binding, which we need to construct the derivation.

### 5.2.5 Lookups in the Environment

At *exprVar*, the goal is to prove that there is a binding in the environment with the right type. The overall idea is that we construct all possible derivations of bindings for an identifier, using the *lookupLam* nonterminal mentioned earlier.

When there is only one possibility, we incorporate it in the substitution, and repeat the visits. The extra type information may rule out other derivations, and result in new type information, etc. Eventually a fixpoint is reached. From all the remaining ambiguous derivations, we pick the deepest ones, and *default* to those, by incorporating their changes into the substitution. We then repeat the process from the beginning, until no ambiguities remain. We run this process on the expression as a whole. In more complex examples that have a let-binding, this process could be repeated per binding group. In the purely functional RulerCore language, we encode this necessarily imperative process using repeated invocation of visits combined with a chained substitution.

We show the implementation of the above algorithm in a step by step fashion. Recall Figure 5.1. Three nonterminals play an essential role: *Lookup* is invoked from the *exprVar* clause and delegates to *LookupMany* to create all derivations possible. To create one derivation, *LookupMany* creates *LookupOne* derivations, one for each nested production *lookupLam* that was put for that identifier into the environment at *exprLam*:

```

sem lookupTy -- invoked from exprVar
  child forest :: LookupMany = lookupMany
  forest.lks = find [] lhs.nm lhs.env -- all LookupOnes
  forest.ty = lhs.ty -- inherited attr of Lookup

```

The *lks* attribute is a list of coroutines. The coroutine *lookupMany* instantiates each of them, and passes on the *ty* attribute to each:

```

itf LookupMany
  visit dispatch inh lks :: [LookupOne]
  inh ty :: Ty

lookupMany = sem lkMany :: LookupMany
sem lkMany
  clause lkNil of dispatch -- when lhs.lks is empty
  clause lkCons of dispatch -- when it has an element
sem lkNil
  match [] = lhs.lks -- reached end of the list
sem lkCons
  match (loc.hd : loc.tl) = lhs.lks
  child hd :: LookupOne = loc.hd -- taken from list
  child tl :: LookupMany = lookupMany -- recurse
  tl.lks = loc.tl -- remainder of the lookups
  default ty -- passes the types by default downwards to hd and tl

```

If all the matching *lookupsOnes* are reduced to one, we pick that one and return its substitution. Otherwise, we return the substitution belonging to the innermost binding (which has

highest *depth*):

```

itf Lookup LookupOne LookupMany
  visit resolve -- hunt for a derivation
    chn subst :: Subst
    syn status :: Status -- outcome of the visit
    syn depth :: Int -- depth of the binding
  visit resolved -- invoked afterward

data Status = Fails | Succeeds { amb :: Bool, change :: Bool }
isAmbiguous (Succeeds True _) = True
isAmbiguous _ = False

```

Every visit is invoked at least once, unless it is declared to be hidden for a child. We intend to invoke the *resolve* visit multiple times. We show further below how this is done.

The depth information is easily determined at the binding-site for lambda expressions, with an inherited attribute *depth*, starting with 0 at the top, and incrementing it with each lambda:

```

itf TransExpr inh depth :: Int
sem transExpr default depth = 0
sem exprLam b.depth = 1 + lhs.depth

```

The default-rule for an inherited attribute optionally takes a Haskell expression (0 in this case), which is only used when there is no parent attribute with the same name.

The rules for *lookupLam* demonstrate the use of the hide-rule:

```

sem lookupLam -- defined inside exprLam above
  child m :: Unify = unify -- try match of binding type
  rename subst1 := subst of m -- to use-site type lhs.ty
  hide outcome of m -- declare not to visit outcome
  m.ty1 = outer.fr.ty -- of enclosing exprLam
  m.ty2 = lhs.ty
  lhs.status = if m.success then Succeeds False m.changes else Fails
  lhs.depth = outer.lhs.depth -- of enclosing exprLam

```

With *hide*, we state not to invoke a visit and the visits that follow. Referencing to attributes of such a visit is considered a static error.

The rules of the *lkCons* clause represent a choice. If from the attributes of the children can be concluded that one derivation remains, it delivers that one's substitution as result. Otherwise, it indicates that an ambiguous choice remains. The lookup with the highest depth is by construction at the beginning of the list:

```

sem lkNil
  lhs.depth = 0 -- lowest depth
  lhs.subst = lhs.subst -- no change to subst
  lhs.status = Fails

```

## 5 Derivation Tree Construction

```

sem lkCons
  hd.subst = lhs.subst -- passed down to
  tl.subst = lhs.subst -- both
  (loc.pick, lhs.status, lhs.depth, lhs.subst)
  = case hd.status of
    Fails → (False, tl.status, tl.depth, tl.subst)
    Success _ hdc →
      let status' = case tl.status of
        Fails → hd.status
        Success _ tlc → Success True (hdc ∨ tlc)
      in (True, status', hd.depth, hd.subst)

```

When a visit is invoked again, we typically want to access some results of a previous invocation. To retain state between multiple invocations of a visits, we allow visits to take visit-local chained attributes. For example, an attribute *decided* for visit *resolve*:

```

sem lookupTy visit.resolve.decided = False -- initial value

```

From inside the visit, we can match on these attributes to select a clause. Furthermore, there is an implicit default rule for them:

```

sem lookupTy
  clause lkRunning of resolve -- no final choice yet,
    match False = visit.decided -- try again
    visit.decided = isAmbiguous lk.status
    default status depth subst -- just pass on
  clause lkFinished of resolve -- made final choice
    match True = visit.decided
    lhs.status = Success False False -- no change
    lhs.depth = 0
    default subst

```

The states of child nodes that are introduced by a previous visit are properly maintained if their visits are also repeated. However, child nodes that are created in a visit are not retained when the visit is repeated. To prevent a created node from being discarded, it is possible to store a node in an attribute. Recall that children are derivations, which are instances of a coroutine, and these are first class values. The detach-rule can exactly be used for this purpose:

```

attr = detach visitname of childname ›

```

Evaluation of a detach-rule takes the child *childname* that is evaluated up to but not including visit *visitname*, and stores it in an attribute *attr*.

A detached child can be attached with an attach-rule:

```

attach visitname of childname = expr

```

The Haskell expression *expr* represents a tree in a state prior to visit *visitname*. If *childname* already exists as child, the attach-rule overrules the visits starting from *visitname*.

The *resolve* visits on *lookupTy* are invoked from *resolve* visits of *transExpr*. In *map deflMap*, we maintain the substitutions of ambiguous lookups per depth. These have not been incorporated in *subst<sub>2</sub>* yet. Applying the deepest of those substitutions *defaults* the choice for the corresponding bindings:

```
itf TransExpr
  visit! resolve
    chn subst2 :: Subst
    syn changes :: Bool -- True iff subst2 was affected
    syn deflMap :: IntMap [Subst] -- defaulting subst/depth
```

The bang at the *resolve* visit indicates that all attributes must be scheduled explicitly to this visit. No attribute is automatically assigned to this visit. This gives the visit a predictable interface, which is convenient when invoking the visit explicitly, as we do further below:

```
sem transExpr
  default changes = or
  default deflMap = unionsWith (++)
  default subst2
```

For ambiguous lookups in the *exprVar*, we add to *deflMap*:

```
sem exprVar
  clause varLkAmb of resolve -- put lk.subst in deflMap
    match (Success True _) = f.status
    lhs.deflMap = singleton lk.depth [lk.subst]
    lhs.subst2 = lhs.subst2 -- bypass lk.subst
  clause varLkOther of resolve -- default rules only
```

To drive the iterations, we introduce a production *iterInner*, which invokes visit *resolve* one or more times. The *iterate*-rule denotes the repeated invocation of a visit of a child:

```
iterate visitname of childname = expr
```

The expression *expr* represents the coroutine of a special production (*iterNext*, explained further below) that computes the inherited attributes for the visit of the next iteration from synthesized attributes of the previous iteration. The iterations stop when this special production does not have an applicable clause:

```
iterInner = sem iterInner :: ExprTrans
sem iterInner
  child e :: ExprTrans = translate -- iterInner is an extra node
  e.expr = lhs.expr -- on top of the derivation tree
  default ... -- omitted: same defaults as transExpr
```

## 5 Derivation Tree Construction

```
iterate resolve of  $e = next$  -- until  $e.changed$  is False  
 $lhs.subst_2 = \mathbf{let}$   $pairs = toAscList\ e.deflMap ++ [(0, [e.subst_2])]$   
                   $substs = head\ pairs$  -- deepest substitutions  
                  in  $foldl\ substMerge\ e.subst_2\ substs$  -- apply them  
 $lhs.changes = \neg (null\ e.deflMap)$ 
```

This special production has as interface the *contravariant* interface of the visit *resolve* of *ExprTrans*, i.e. the inherited attributes turn to synthesized attributes, and vice versa. The triple instead of dual colons indicate this difference:

```
 $next = \mathbf{sem}\ iterNext :: ExprTrans.resolve$  -- one anonymous clause  
match  $True = lhs.changes$  -- stops when there are no changes  
default  $subst_2$  -- pass prev  $subst_2$  into the next iter
```

Finally, we introduce a production *wrapper*, which forms the root of the derivation tree and invokes the visits on the derivation for expressions, including again an iteration of the inner loop:

```
itf Compile inh expr :: ExprS  
      inh  $env :: Env$   
      syn  $subst :: Subst$   
      syn  $ty :: Ty$   
 $compile = \mathbf{sem}\ wrapper :: Compile$   
sem  $wrapper$   
      child  $e :: TransExpr = iterInner$   
      default  $env\ expr\ ty$   
      iterate resolve of  $e = next$  -- repeat the inner loop  
       $lhs.subst = e.subst_2$ 
```

### 5.2.6 Translation to the Target Expression

The code so far computes the information needed to translate the source expression. The shape of the derivation is determined, and after iterations, *subst<sub>2</sub>* contains the substitution for the types. We wrap up with generating the target expression as attribute *trans* and collecting the errors:

```
itf ExprTrans visit generate  
      inh  $subst_3 :: Subst$   
      syn  $trans :: ExprT$   
      syn  $errs :: Errs$   
sem  $exprVar\ lhs.trans = VarT\ lk.nm'$  --  $lk$  delivers the name  
sem  $exprApp\ lhs.trans = AppT\ f.trans\ a.trans$   
sem  $exprLam\ lhs.trans = LamT\ loc.u\ b.trans$   
sem  $exprDeriv$  default  $errs = concat$ 
```



```

itf Compile syn trans :: ExprT
sem wrapper default trans
      e.subst3 = e.subst2

```

The lookupTy nonterminal delivers the name for a variable. The alternatives were constructed in iterations of the *resolve* visits, and stored in the *loc.mbDeriv* attribute. We take it out and continue from there. From the derivations of nonterminal *lkMany*, we pick the name for the first one that has *loc.pick* equal to *True*:

```

itf Lookup visit resolved
      syn nm' :: Ident
      syn errs :: Errs

sem lookupTy
      lhs.errs = maybe [Err_unresolved lhs.nm] (const []) lk.mbNm
      lhs.nm'  = maybe lhs.nm id lk.mbNm

itf LookupMany
      syn mbNm :: Maybe Ident

sem lkNil  lhs.mbNm = Nothing
sem lkCons lhs.mbNm = if loc.pick then Just hd.nm' else tl.mbNm

itf LookupOne syn nm' :: Ident      -- use u as name
sem lookupLam lhs.nm' = outer.loc.u -- defined in exprLam

```

The remaining code of the compiler invokes the coroutine generated from the *compile* production. It provides the *ExprS* expressions, and obtains the type and an *ExprT* back. We omit these details.

### 5.2.7 Discussion

**Performance.** Clauses introduce backtracking. In the worst case, this leads to a number of traversals that are exponential in the size of the (longest intermediate) tree. In practice, clause selection is typically a function of some inherited attributes (i.e. deterministic), which only requires a constant number of traversals over the tree. For example, this is the case for RulerCore programs expressible in UUAG. We verified that programs generated from RulerCore exhibit the same time and memory behavior as programs generated from UUAG.

**Expressiveness.** With attributes, we conveniently compute information in one part of the tree, and transport the information to other parts, which allows context-dependent decisions to be made. The notion of visits gives us sufficient control to steer the inference process.

On the other hand, it is not possible to simply plug a type system in RulerCore and automatically obtain an inference algorithm. We provide the building blocks to write inference algorithms for many type systems, but it is up to the programmer to ensure that the result is sound and complete.

Soundness of a RulerCore program is typically easy to prove. Completeness, however, is a different issue. That largely depends on decisions made about unknown types. With

## 5 Derivation Tree Construction

RulerCore, we make explicit when choices are made, and when visits are repeated. We believe this helps when reasoning about completeness.

**Constraint-based inference.** We establish the following relation to constraint-based inference: a detached derivation can be seen as a constraint, can be collected in an attribute and solved elsewhere. Solving constraints corresponds invoking visits (such as *resolve* in Section 5.2.5) on the derivation, potentially multiple times.

Solving a constraint may result in more constraints. We store these either in a node's state, or collect them in attributes.

A constraint is typically parametrized with information from the context that created it. We provide access to this context via nested nonterminals, which have access to the attributes of their outer nonterminals.

## 5.3 Semantics

We define RulerBack, a small core language for Attribute Grammars. We translate a RulerCore program in two steps to Haskell. We first desugar RulerCore. This gives us a RulerBack program. We then translate the latter to Haskell. The separately defined attributes of RulerCore are grouped together in RulerBack, visits are ordered, attributes allocated to visits, covariant interfaces translated to normal interfaces, rules ordered based on their attribute dependencies, and rules augmented with default rules. We omit description of this translation, as it is similar to the frontend of UUAG [Löh et al., 1998]. Instead, we focus on the translation to Haskell, which precisely defines the semantics of RulerBack, and thus forms the underlying semantics of RulerCore.

### 5.3.1 Syntax

The RulerBack language is Haskell extended with additional syntax for toplevel interface declarations, semantic expressions, and attribute occurrence expressions. The following grammar lists these syntax extensions:

```
 $i ::= \mathbf{if} I \bar{v}$  -- interface decl, with visits  $v$   
 $v ::= \mathbf{visit} x \mathbf{inh} \bar{a}_1 \mathbf{syn} \bar{a}_2$  -- visit decl, with attributes  $a_1$  and  $a_2$   
 $a ::= x :: \tau$  -- attribute decl, with Haskell type  $\tau$   
 $s ::= \mathbf{sem} x :: I t$  -- semantics expr, defines production  $x$   
 $t ::= \mathbf{visit} x_1 \mathbf{chn} \bar{x}_2 \bar{r} \bar{c}$  -- visit def, with common rules  $r$   
|  $\square$  -- end of the visit sequence  
 $c ::= \mathbf{clause} x \bar{r} t$  -- clause definition, with next visit  $t$   
 $r ::= p \leftarrow e$  -- assert-rule, evaluates monadic  $e$   
|  $\mathbf{match} p \leftarrow e$  -- match-rule, backtracking variant  
|  $\mathbf{invoke} x \mathbf{of} c \leftarrow e$  -- invoke-rule, invokes  $x$  on  $c$ , while  $e$   
|  $\mathbf{attach} x \mathbf{of} c :: I \leftarrow e$  -- attach-rule, attaches a partially evaluated child  
|  $p = \mathbf{detach} x \mathbf{of} c$  -- detach-rule, stores a child in an attr
```

$o ::= x.x$  -- expression, attribute occurrence  
 $x, l, p, e$  -- identifiers, patterns, expressions respectively

There are some differences in comparison with the examples of the previous section. Invocations of visits to children are made explicit through the *invoke*-rule, which also represents the *iterate*-rule. Similarly, the *attach* rule also takes care of introducing children. A visit definition declares number of visit-local chained attributes  $\bar{y}$ , and has a number of rules to be evaluated prior to the evaluation of clauses. A clause defines the next visit, if any.

The order of appearance of rules determines the evaluation order, which allows them to be monadic. Non-monadic expressions are lifted with *return*. The implementation is parametrizable over any backtracking monad. In this chapter, we use *IO* as example.

### 5.3.2 Example

The following example is taken from Section 3.3. It demonstrates how to compute the sum of a list of integers in two visits in RulerBack. In the first visit, the attribute  $l$  is inspected to obtain the elements in the list. In the second visit, the elements are summed up:

```

if  $S$  visit  $v_1$  inh  $l :: [Int]$  syn  $\emptyset$  -- decompose list  $l$  down
  visit  $v_2$  inh  $\emptyset$  syn  $s :: Int$  -- compute sum  $s$  up
 $sum' = \mathbf{sem}$   $sum :: S$ 
  visit  $v_1$  chn  $\emptyset \emptyset$ 
    clause  $sumNil_1$  -- when list is empty
      match  $[] \leftarrow \mathbf{return}$   $lhs.l$  -- match  $[] = l$ 
      visit  $v_2$  chn  $\emptyset \emptyset$  -- no visit-local attrs
      clause  $sumNil_2$ 
         $lhs.s \leftarrow \mathbf{return}$   $0$  -- empty list, zero sum
         $\square$  -- no next visit
    clause  $sumCons_1$  -- when list non-empty
      match  $(loc.x : loc.xs) \leftarrow \mathbf{return}$   $lhs.l$  -- match  $(x : xs) = l$ 
      attach  $v_1$  of  $tl :: S \leftarrow \mathbf{return}$   $sum$  -- recursive call
       $tl.l \leftarrow \mathbf{return}$   $loc.xs$  --  $l$  param of call
      invoke  $v_1$  of  $tl \leftarrow \mathbf{noIteration}_S$  -- visit it to pass  $l$ 
      visit  $v_2$  chn  $\emptyset \emptyset$ 
        clause  $sumCons_2$ 
          invoke  $v_2$  of  $tl \leftarrow \mathbf{noIteration}_S$  -- visit it to get the sum
           $lhs.s \leftarrow \mathbf{return}$   $(loc.x + tl.x)$  -- sum of  $hd$  and the  $tl$ 
           $\square$  -- no next visit

```

We translate a RulerBack production to a coroutine, in the form of continuations. From the interface, we generate a type signature for these coroutines:

```

type  $S$  =  $S_{v_1}$ 
newtype  $S_{v_1} = S_{v_1} ([Int] \rightarrow IO ((), (S_{v_1}, S_{v_2})))$ 
newtype  $S_{v_2} = S_{v_2} (() \rightarrow IO (Int, (S_{v_2}, \square)))$ 

```

## 5 Derivation Tree Construction

Inherited attributes become parameters, and synthesized attributes are returned as a tuple of results. Each visit also returns two continuations of type  $S_{v_1}$  and  $S_{v_2}$  respectively. The first continuation represents the current visit itself (which may be re-invoked with updated internal state), the second continuation represents the next visit, or  $\square$  if there is no subsequent visit. Since no inherited attributes have been declared for the second visit, the continuation of type  $S_{v_2}$  can actually be represented as a value:

**newtype**  $S_{v_2} = S_{v_2} (IO (Int, (S_{v_2}, \square)))$

The coroutine  $sum'$  has  $S$  as type. Attributes are encoded as a variable *childAttr* or *childOAttr*, depending on whether the attribute is an input or output of the clause. Clause selection relies on backtracking in the monad. When a match-statement doesn't match, a failure is generated in the monad, which we *catch* to switch to the next clause.

```

sum' = S_v1 vis_v1 where
  vis_v1 lhs_l = (
    do [] ← return lhs_l
    let r = S_v1 vis_v1
        k = S_v2 vis_v2 where
          vis_v2 = (
            do lhs_os ← return 0
                let r = S_v2 vis_v2
                    k = □
                    return (lhs_os, (r, k))
            ) 'catch' (\_ → ⊥)
          return ((, (r, k))
        ) 'catch' (\_ →
    do (loc_Lx : loc_Lxs) ← return lhs_l
        tl_ol ← return loc_Lxs
        (S_v1 vis_tl_v1) ← return sum'
        ((, (-, S_v2 vis_tl_v2)) ← vis_tl_v1 tl_ol
    let r = S_v1 vis_v1
        k = S_v2 vis_v2 where
          vis_v2 = (
            do (tl_s, (-, -)) ← vis_tl_v2
                lhs_os ← return (loc_Lx + tl_s)
                let r = S_v2 vis_v2
                    k = □
                    return (lhs_os, (r, k))
            ) 'catch' (\_ → ⊥)
          return ((, (r, k)))
  ) 'catch' (\_ →
-- the initial state
-- first clause of visit v1
-- match on lhs.l
-- repetition cont.
-- next visit cont.
-- clause of visit v2
-- lhs.s computation
-- repetition
-- no next visit
-- deliver result v2
-- no other clause for v2
-- deliver result of visit v1
-- second clause
-- match on lhs.l
-- inherited attr tl.l
-- attach child tl
-- first visit on tl
-- repetition cont.
-- next visit cont.
-- clause of visit v2
-- second visit on tl
-- lhs.s
-- repetition
-- no next visit
-- deliver result v2
-- no other clause for v2
-- deliver result of visit v1

```

The above code is slightly simplified. Below, we show the general translation.

### 5.3.3 Translation

We use the following naming conventions from RulerCore names to Haskell names. The right-hand sides of these definitions consist of string concatenations:

```

outp "loc" x = "locL" x      inp "loc" x = "locL" x
outp "lhs" x = "lhsS" x      inp "lhs" x = "lhsI" x
outp c      x = c "I" x      inp c      x = c "S" x
outp y      = "visitS" y     inp      y = "visitI" y
vis c x     = "vis_" c "_" x prod x     = "sem_" x
vis x      = "vis_" x       ity I x = I "_" x
s I x      i I x -- respectively, inh and syn attrs of x of I

```

Note that an attribute  $c.x$  for some child  $x$  at an *output position* represents the inherited attribute  $x$  of  $c$ , and vice versa for attributes at input positions.

The types of the coroutines are generated from an interface declaration:

```

[[itf I  $\bar{v}$ ]]          ~ type [[I]] = [[ity I x'];  $\bar{v}$ ]_I -- x' next visit,
[[visit x inh  $\bar{a}$  syn  $\bar{b}$ ]]_I ~ newtype [[ity I x]] =      -- otherwise ()
  [[ity I x]] ([[ $\bar{a}$ ] → IO ([[ $\bar{b}$ ], ([[ity I x], [[ity I x']]])

```

From these interfaces, we actually also generate wrappers to interface with the coroutines from Haskell code. The translations for them bear a close resemblance to the translation of the attach and invoke rules below.

The clauses of a visit are translated to a function  $[[vis\ x]]$  that tries the clauses one by one. This function takes as parameters the coroutines  $[[chlds]]$  of the children in scope prior to invoking the visit, the visit-local attributes  $\bar{y}$ , and the inherited attributes:

```

[[sem x :: I t]]      ~ let [[prod x]] = [[t]]_I in [[prod x]]
[[()]]_I              ~ ()
[[visit x chn  $\bar{y}$   $\bar{r}$   $\bar{c}$ ]]_I ~
  let [[vis x]] [[chlds]] [[inp  $\bar{y}$ ]] [[inp lhs (i I x)]]
    = catch (do { [[ $\bar{r}$ ]]; [[ $\bar{c}$ ]]_I,x, $\bar{y}$  }) ⊥ in [[ity I x]] [[vis x]]

```

The clauses themselves translate to a sequence of statements, consisting of the translated statements of the semantic rules, and the construction of the two continuations. We partially parametrize both continuations with the updated children:

```

[[[]]]_I,v, $\bar{y}$           ~ error "no clause applies"
[[clause x  $\bar{r}$  t : cs]]_I,v, $\bar{y}$  ~
  catch (do { [[ $\bar{r}$ ]]; let { [[inp x  $\bar{y}$  = outp  $\bar{y}$ ]] }
    ; let { r = [[ity I x]] [[vis x]] [[chlds]] [[outp x  $\bar{y}$ ]]
      ; k = [[t]]_I,chlds }
    ; return ([[outp lhs (s I x)], (r, k) ]})
  (\_ → [[cs]]_I,v, $\bar{y}$ )

```

## 5 Derivation Tree Construction

$$\begin{aligned} \llbracket () \rrbracket_{I,ks} & \rightsquigarrow () \\ \llbracket \mathbf{visit} \ x \ \mathbf{chn} \ \bar{y} \ \bar{r} \ \bar{c} \rrbracket_{I,ks} & \rightsquigarrow \llbracket \mathbf{visit} \ x \ \mathbf{chn} \ \bar{y} \ \bar{r} \ \bar{c} \rrbracket_I \llbracket ks \rrbracket \llbracket \mathit{outp} \ x \ \bar{y} \rrbracket \end{aligned}$$

Semantic rules translate to monadic statements. For the assert-rule, we match using a let-statement, which ensures that a pattern match failure is considered a runtime error, instead of cause backtracking in the monad:

$$\begin{aligned} \llbracket \mathbf{match} \ p \leftarrow e \rrbracket & \rightsquigarrow \llbracket p \rrbracket \leftarrow \llbracket e \rrbracket \\ \llbracket p \leftarrow e \rrbracket & \rightsquigarrow x \leftarrow \llbracket e \rrbracket; \mathbf{let} \ \{ \llbracket p \rrbracket = x \} \quad \text{-- } x \text{ fresh} \\ \llbracket \mathbf{attach} \ x \ \mathbf{of} \ c :: I \leftarrow e \rrbracket & \rightsquigarrow (\llbracket \mathit{ity} \ I \ x \rrbracket \llbracket \mathit{vis} \ c \ x \rrbracket) \leftarrow \llbracket e \rrbracket \\ \llbracket p = \mathbf{detach} \ x \ \mathbf{of} \ c \rrbracket & \rightsquigarrow \mathbf{let} \ \{ \llbracket p \rrbracket = \llbracket \mathit{ity} \ I \ x \rrbracket \llbracket \mathit{vis} \ c \ x \rrbracket \} \end{aligned}$$

Invoke invokes a visit  $x$  (named  $f$  in the translation) on child  $c$  once, then repeats invoking it, as long as  $e$  (named  $g$ ) succeeds in feeding it new input:

$$\begin{aligned} \llbracket \mathbf{invoke} \ x \ \mathbf{of} \ c \leftarrow e \rrbracket & \rightsquigarrow \\ & (\llbracket \mathit{inp} \ c \ (s \ I_c \ x) \rrbracket, (-, k)) \\ & \leftarrow \mathbf{let} \ \mathit{iter} \ f \ \llbracket \mathit{outp} \ c \ (i \ I_c \ x) \rrbracket = \mathbf{do} \\ & \quad \{ (\llbracket \mathit{coItty} \ I_c \ x \rrbracket \ g) \leftarrow \llbracket e \rrbracket \\ & \quad ; z @ (\llbracket \mathit{inp} \ c \ (s \ I_c \ x) \rrbracket, (\llbracket \mathit{ity} \ I_c \ x \rrbracket f', -)) \\ & \quad \quad \leftarrow f \ \llbracket \mathit{outp} \ c \ (i \ I_c \ x) \rrbracket \\ & \quad ; \mathbf{catch} \ (\mathbf{do} \ \{ (\llbracket \mathit{outp} \ c \ (i \ I_c \ x) \rrbracket, -) \leftarrow g \ \llbracket \mathit{inp} \ c \ (s \ I_c \ x) \rrbracket \\ & \quad \quad ; \mathit{iter} \ f' \ \llbracket \mathit{outp} \ c \ (i \ I_c \ x) \rrbracket \}) \\ & \quad \quad (\backslash_- \rightarrow \mathbf{return} \ z) \} \\ & \quad \mathbf{in} \ \mathit{iter} \ \llbracket \mathit{vis} \ c \ x \rrbracket \ (\mathit{outp} \ c \ (i \ I_c \ x)) \\ & ; \mathbf{let} \ (\llbracket \mathit{ity} \ I_c \ x' \rrbracket \llbracket \mathit{vis} \ c \ x' \rrbracket) = k \quad \text{-- } x' \text{ is next visit, or line omitted} \end{aligned}$$

Finally, we add bangs around patterns to enforce evaluation, and replace attribute occurrences with their Haskell names:

$$\begin{aligned} \llbracket C \ \bar{p} \rrbracket & \rightsquigarrow !(C \ \overline{\llbracket p \rrbracket}) \\ \llbracket (p, \dots, q) \rrbracket & \rightsquigarrow !(\llbracket p \rrbracket, \dots, \llbracket q \rrbracket) \\ \llbracket c.x \rrbracket & \rightsquigarrow !\llbracket \mathit{inp} \ c \ x \rrbracket \\ \llbracket e \rrbracket & \rightsquigarrow e \ [c.x := \llbracket \mathit{outp} \ c \ x \rrbracket] \end{aligned}$$

The translation exhibits a number of properties. If the RulerCore or RulerBack program is well typed, then so is the generated Haskell program, and vice versa. Furthermore, the translation is not limited to Haskell. A translation similar to above can be given for any language that supports closures.

## 5.4 Related Work

Attribute grammars as defined by Knuth [1968] are extensions of context free grammars. Typically, an attribute grammar is defined in terms of a context-free abstract grammar of the language to analyze or compile. The attribute evaluator computes attributes of the abstract

syntax tree that is determined apriori by a parser. In case of type inference, when the typing relations are not directed by syntax, the derivation tree is not known beforehand. Thus, the derivation tree cannot be expressed directly in terms of attribute grammars, unless higher-order attributes are used [Vogt et al., 1989].

**Tree manipulations.** There are many extensions to attribute grammars to facilitate changing the tree during attribute evaluation. Silver [Van Wyk et al., 2008], JastAdd [Ekman and Hedin, 2007] and UUAG [Löh et al., 1998] support higher-order attributes. These grammars allow the tree to be extended with subtrees that are computed from attributes, and subsequently decorated. The responsibility of selecting a production of a higher-order child lies with the parent of that child, and the choice is final. In RulerCore, a child itself selects a clause to make a choice, and a choice can be made per visit.

JastAdd and Aster [Kats et al., 2009], support conditional rewrite rules, which allows rigorous changes to be made to the tree. Coordination between rewriting and attribute evaluation is difficult to express due to mutual influence, especially if the transformations are not confluent. To limit interplay, JastAdd’s rewriting of a tree is limited to the first access of that tree.

Many type inference algorithms, especially for type and effect systems, iteratively traverse the tree. Some algorithms construct additional subtrees during this process. Circular Attribute Grammars [Jones, 1990], supported by JastAdd and Aster, iteratively compute circular attributes until a fixpoint is reached. UUAG and Silver can deal with circularity via lazy evaluation with streams. CAGs, however, do not support changes to the tree during these iterations. Stratego’s rewrite mechanism that underlies Aster, however, is more general and can change the tree. In RulerCore, a visit may be iterated several times. Each node in the derivation tree can maintain a per-visit state to keep track of newly constructed parts of the tree.

**Non-deterministic trees.** The attribute grammar systems above have in common that they massage a tree until it has the right form. Alternatively, a tree can be constructed non-deterministically, using e.g. logic programming languages. The grammar produces only the empty string, and the semantic rules disambiguate the choice of productions. Arbab [1986] show how to translate attribute grammars to Prolog. However, this approach does not allow the inspection of partial *LookupOne* derivations of Section 5.2.5, nor the defaulting, to be implemented easily. With RulerCore, we offer alternative constructions of the tree per visit in combination with backtracking. The notion of a visit provides an intuitive alternative for the *cut* operator.

Prolog-like approaches also offer unification mechanisms to deal with non-determinism in attribute computations. In contrast, we require the programmer to either program unifications and substitutions manually, or use a logic monad combined with a unification in the translation of Section 5.3.

Engelfriet and Filé [1989] show the expressiveness of classes of attribute grammars. Unsurprisingly, deterministic AG evaluators have lower computational complexity bounds compared to non-deterministic ones. With RulerCore, we target large compilers (i.e. UHC), that

## 5 Derivation Tree Construction

processes large abstract syntax trees, thus we need the control on the exploration of alternatives that visits offer.

**Related attribute grammar techniques.** Several attribute grammar techniques are important to our work. Kastens [1980] introduces ordered attribute grammars. In OAGs, the evaluation order of attribute computations as well as attribute lifetime can be determined statically, allowing severe optimizations.

Boyland [1996] introduces conditional attribute grammars. In such a grammar, semantic rules may be guarded. Our clauses-per-visit model also provide guarded rules, but in addition also allow children to be conditionally defined.

Saraiva and Swierstra [1999, chap. 3] describe multi-traversal functions in a functional language (or visit functions [Swierstra and Alcocer, 1998]). These visit functions are one-shot continuations, or coroutines without looping. We improved upon this mechanism to support iterative invocation of visits, thus encoding coroutines with loops.

## 5.5 Conclusion

We presented the extensions that RulerCore, a conservative extension of ordered attribute grammars, provides to describe type inference algorithms. We explained RulerCore with an extensive example in Section 5.2 and described its semantics in Section 5.3.

RulerCore has several distinct features. Firstly, in contrast to most attribute grammar systems, construction of a derivation tree and the evaluation of its attributes is intertwined in RulerCore. This allows us to define a grammar for the language of derivations of some typing relations, instead of being limited to the grammar of expressions or types.

Secondly, we use the notion of explicit visits to capture the gradual, effectful nature of type inference. Each visit corresponds to a state transition of the derivation tree under construction. These visits may be repeated to form fixpoint iterations.

Thirdly, many inference algorithms reason about what part of the derivation is known, or is still pending, e.g. by means of constraints. In RulerCore, derivation trees are first class and can be inspected by visiting them, which facilitates such reasoning in terms of attributed trees.



## 6 AGs with Stepwise Evaluation

Attribute Grammars are a powerful formalism to specify and implement the semantics of programming languages (e.g. as in a compiler), in particular when the semantics are syntax directed. Advanced type systems, however, have declarative specifications that encode decisions that are independent of syntax. The implementation of such decisions is hard to express algorithmically using conventional attribute evaluation.

This chapter presents Stepwise Attribute Grammars (SAGs). In a SAG, nondeterministic choices can be expressed in a natural way in conjunction with unambiguous resolution strategies based on attribute values. SAGs preserve the functional relationships between attributes and support on-demand evaluation. The exploration of alternatives are encoded as a choice between the semantic results of children. Evaluation of a child can be performed in a stepwise fashion: it is paused after each step and yields a progress report with intermediate results, until the child is reduced to its semantic value. This facilitates a breadth-first exploration of choices, until choices can be resolved based on the progress reports.

### 6.1 Introduction

Attribute Grammars (AGs) [Knuth, 1968] are a formalism that is particularly suited for the concise implementations for semantics of programming languages (e.g. static, operational, denotational), in the form of a compiler or interpreter. Hereby, attributes play a crucial role: properties of Abstract Syntax Trees (ASTs) can easily be expressed as attributes, and combined to form more complex properties. These attributes can be shared and additional attributes can be added on demand. For example, attributes related to name analysis and type checking can be used in a later stage for code generation or the collection of error messages. We used AGs for many small, but also several large projects, including the Utrecht Haskell Compiler (UHC) [Dijkstra et al., 2009], the Helium compiler for learning Haskell, the Generic Haskell Compiler, and the editor Proxima. AGs, and corresponding tool support, proved to be essential for these projects.

Modern programming languages allow a compiler to take some of the implementation effort away from the programmer. In C#, local type inference is employed, such that an abundance of type signatures can be omitted. Many typed OO languages have an auto-boxing feature to automatically wrap primitive values into objects. Such features save programmers from tedious tasks. To specify this freedom, programming language specification are declarative, and impose sufficient restrictions such that a deterministic algorithm exists.

Unfortunately, it is hard to deal with declarative type systems using conventional AGs. For example, it is not immediately obvious how to express Haskell's overloading for UHC using an AG. In fact, we currently rely on a constraint solver external to UHC's AGs. A lot of boilerplate code is required to interface with such a solver, and it introduces an artificial

phase distinction (which hampers on-demand evaluation). Consequently, it increases code complexity severely. The goal of this chapter is therefore to extend AGs such that inference algorithms for declarative type systems can be expressed in a natural way.

Since an AG is both a functional specification and *implementation*, our challenge is to describe algorithms that resolve declarative aspects. The declarative aspects in a semantics occurs in two forms: attributes with a non-functional definition (e.g. fresh types), and productions that are not syntax-directed, but where their applicability depends on values of attributes. These two aspects are mutually expressible (Appendix G). As we prefer attributes to be functionally defined, we focus on the second notion. Typically, we can deal with declarative aspects using a unification-based approach, which integrates well with AGs [Dijkstra and Swierstra, 2004]. However, some applications require a more powerful approach, with an algorithm that *actively* tries out alternatives at choice points. For example, in a Haskell compiler, to search for an equality proof for GADTs, and to resolve overloading in the presence of undecidable instances. Unfortunately, *exploration of alternatives* does not fit AGs straightforwardly.

Such explorations do not fit out of the box, because productions are selected based on the syntax (e.g. the parsed AST), and not on the values of attributes. To lift this restriction, there are approaches that generate Prolog [Walsteijn and Kuiper, 1986, Arbab, 1986]. However, we have several additional demands:

1. The approach needs to be compatible with any general purpose host language. From a theoretical perspective: to allow the extension we propose to be integrated in other AG systems as well; from a practical perspective: we have a large Haskell code base, thus want to use Haskell as a host language.
2. We need a *breadth-first evaluation* with *online results* to deal with potential infinite ASTs. Online results are also needed for integration with conventional on-demand evaluation.
3. We need complex disambiguation strategies. In general, for deterministic results and error reporting; in particular, to deal with some extensions to type classes that UHC offers [Dijkstra et al., 2007].

In this chapter, we present an approach that adds exploration of alternatives to AGs and meets the above demands.

Our approach consists of following three key ideas:

1. We encode declarative aspects and their resolution as a *choice function* between children of a production. Overall, attributes are evaluated using conventional on-demand evaluation. Attributes of children participating in a *choice*, however, are evaluated *strictly*.
2. We annotate productions such that they *yield* user-defined progress reports that contain *intermediate results* upon strict attribute evaluation. After yielding a progress report, evaluation for a child *pauses*. This allows the choice function to evaluate its children in a *step-by-step fashion* until it has observed sufficient progress reports to commit to one of the children. It may yield progress reports itself in the mean time.

3. As implementation strategy, we map each production to a special form of coroutine that can both be run greedily until it yields the next progress report, and be run to completion directly but with a lazy result. This implementation facilitates a *hybrid evaluation model* between stepwise and on-demand evaluation, and is therefore compatible with other AG techniques that depend on the conventional on-demand evaluation.

We motivate these ideas via an example in Section 6.2.

These ideas are inspired by a parsing technique by Swierstra [2009], Hughes and Swierstra [2003] to explore alternatives based on progress information reported by parsers. The approach in this chapter is not directly related to parsing, but ultimately has its roots in the above technique (for a detailed comparison, cf. Section 6.7).

In the text, we refer to appendices published in an accompanying technical report [Middelkoop et al., 2010e] where certain topics are explored in more detail. The main contributions of this chapter are:

- We define SAGs, a language extension to Attribute Grammars that copes with declarative aspects in a semantics, while keeping the AG purely functional. We explain SAGs in Section 6.2, and show how to translate them to Haskell as a host language in in Section 6.3. We implemented this extension in the UUAG system [Löh et al., 1998].
- We introduce lazy coroutines, or *stepwise computations*, for which we provide an reference implementation (Section 6.4). In this chapter, we focus on the main ideas. The Haskell library<sup>1</sup> shows and explains many additional features that are useful in practice (see also Section 6.6). The library may also be used by Haskell programs that need powerful exploration capabilities, but are not related to AGs.
- As a proof of concept that these techniques are portable to other languages as well, we also provide an implementation of the example in Section 6.2 in Java (Section 6.5).

## 6.2 Example of a Stepwise AG for a Predicate Language

In this section, we take as running example an operational semantics<sup>2</sup> for a Boolean predicate language *Pred*. We show how to write *Pred*'s semantics as an AG using the notation as supported by the UUAG system. The semantics is executable: its implementation yields an interpreter in Haskell for *Pred*. Initially, the example is a functional specification, such that we can resort to a conventional AG and explain the notation involved [Löh et al., 1998]. Next we show that the semantics has an efficiency problem which we can solve by making the specification more declarative. With a Stepwise AG we then deal with it.

---

<sup>1</sup> Stepwise monad: <https://svn.science.uu.nl/repos/project.ruler.papers/archive/stepwise-1.0.2.tar.gz>

<sup>2</sup> For an example related to type systems, see Appendix F.

### 6.2.1 Syntax of the Predicate Language

Consider the following grammar for the abstract syntax of *Pred*:

```

data Pred -- Grammar for nonterm Pred as algebraic data type.
  | Var nm :: String      -- Variable with value in Env.
  | Let nm :: String      -- Binds (non-recursively) a
    expr :: Pred          -- name to the value of expr,
    body :: Pred          -- in scope of body.
  | And left :: Pred      -- The logical  $\wedge$  of two preds.
    right :: Pred
  | Or left  :: Pred      -- The logical  $\vee$  of two preds.
    right :: Pred

type Env = Map String Bool -- Environment that maps names to Booleans.

```

The data declaration introduces a nonterminal *Pred*, and a number of productions *Var*, *Let*, etc. The fields of the production comprise the symbols of the RHS of the production, consisting of a name *nm*, *expr*, etc. and a type. Some built-in types such as *Bool* and *String* specify that the symbol is a terminal, otherwise the symbol is a nonterminal.

From the grammar, we generate constructor functions *sem\_Var*, *sem\_Let*, etc. which are used to build attributed ASTs. Given an initial environment  $\{ "f" \rightarrow \textit{False}, "t" \rightarrow \textit{True} \}$ , which binds a truth value to the free variables in the predicates, we turn a predicate into a proposition. For example:

```

taut  = sem_Or (sem_Var "t") (sem_Var "f")    -- True
contr = sem_And (sem_Var "t") (sem_Var "f")  -- False
alias = sem_Let "a" (sem_Var "t") (sem_Var "a") -- True
big1 = sem_And (sem_Var "f") big1           -- False
big2 = sem_And big2 (sem_Var "f")           -- False

```

Informally speaking, the Boolean value of *taut* is *True* and of *contr* is *False*. The *big<sub>1</sub>* and *big<sub>2</sub>* predicates are large sequences of *False* that are combined with *ands*. In fact, these sequences are infinitely long, although we only use that for emphasis. Their truth value is *False*. Formally, however, we have to define an operational semantics to make such statements.

### 6.2.2 Deterministic Operational Semantics

An operational semantics for a predicate takes an environment and provides a truth value. We model these two aspects as attributes, which correspond to values attached to the nodes of an AST: an *inherited* attribute *env* that represents the environment for the subtree, and a *synthesized* attribute *val* that denotes the value of the subtree (for the given environment):

```

attr Pred inh env :: Env syn val :: Bool -- Typed attributes for nonterm Pred.

```

The obligation to define an inherited attribute for a node in the AST lies with the parent, for a synthesized attribute it lies with a child. Via a **sem**-block, we define for each production,

## 6.2 Example of a Stepwise AG for a Predicate Language

the production's synthesized attributes, and the inherited attributes of its children using rules written in the AG's host language (in our case: Haskell expressions). These rules may refer to the inherited attributes of the production and the synthesized attributes of the children. We refer to an attribute using *chld.atname* notation. To refer to the attributes of the production itself, we use the special name *lhs*. We refer to terminals by their name. Thus, we define the semantics for predicates as:

```
sem Pred -- Specifies rules for productions of nonterm Pred.
| Var lhs.val = lookup nm lhs.env
| Let lhs.val = body.val -- Takes val from child body.
   body.env = insert nm expr.val lhs.env
| And lhs.val = left.val  $\wedge$  right.val
   left.env = lhs.env -- Copies down env to the left.
   right.env = lhs.env -- Copies down env to the right.
| Or lhs.val = left.val  $\vee$  right.val
   left.env = lhs.env
   right.env = lhs.env
```

Thus, we simply pass down the environment from top to bottom. For a variable, we lookup the associated value in the environment. For a let-binding, we insert the value in the environment. Finally, for the *And* and *Or*, we take the Haskell (short-circuiting) ( $\wedge$ ) and ( $\vee$ ) on Boolean values of the children.

From this AG, the UUAG compiler generates an interpreter that takes a predicate, defines the root's *env* attribute, and demands a value for root's *val* attribute. On-demand evaluation proceeds to compute those attributes when their values are needed during the computation of the *val* attribute. In a purely functional language such as Haskell, we can represent a decorated tree as a function from inherited to synthesized attributes [Saraiva and Swierstra, 1999]. More precisely, for each production (e.g. *Var*, *And*), we generate a function (*sem\_Var*, *sem\_And*, respectively) that, given the functions corresponding to its children, represents a Haskell function takes values for inherited attributes as parameter, and returns a product with values for the synthesized attributes (Section 6.3).

When we run the interpreter on the examples given earlier, it gives the expected outcomes, with a single exception: the computation for *big<sub>2</sub>* diverges. The reason is that both ( $\vee$ ) and ( $\wedge$ ) start with the left argument first. This may involve a lot of computation (e.g. in case of *big<sub>2</sub>*) that could be avoided by looking at the second argument first. Then, however, *big<sub>1</sub>*'s evaluation would diverge. If *big<sub>2</sub>* would be a long but finite sequence, then its evaluation would not diverge but take a long time to produce an answer.

### 6.2.3 Declarative Operational Semantics

The  $\wedge$  and  $\vee$  operators do not distribute evaluation over their operands well. To give more freedom with respect to the evaluation order, we could add non-determinism to the specification, for example via multiple (conditional) interpretations of a production, using the following (fictional) notation:

## 6 AGs with Stepwise Evaluation

```
sem Pred -- Productions with conditional alternatives.
| And1 lhs.val = left.val   when left.val = False
| And2 lhs.val = right.val  when right.val = False
| And3 lhs.val = left.val   when otherwise
| Or1  lhs.val = left.val   when left.val = True
| Or2  lhs.val = right.val  when right.val = True
| Or3  lhs.val = left.val   when otherwise
```

In this specification, there is freedom in the alternative to apply. A clever AG evaluator could use a breadth-first exploration of alternatives combined with prioritizing those attributes that are closer to the root and thus provide a more balanced evaluation strategy.

However, we want to be precise in this strategy. When both *left.val* and *right.val* are *True*, the choice between *Or*<sub>1</sub> and *Or*<sub>2</sub> is ambiguous. For predictability reasons (and *referential transparency*), it must be clear which one should be taken; also, we may prefer one over the other based on some other available information. In some cases we may want to be biased towards a particular subtree, potentially based on results computed at runtime. Hence, as mentioned in Section 6.1, we want to be able to define this strategy ourselves.

### 6.2.4 Stepwise Operational Semantics

To define such a strategy, we take a less ad-hoc solution. In particular, we encode alternatives as a choice between children instead of productions. We do not lose expressiveness (Appendix E), and gain the ability to define strategies in terms of the evaluation of children. In the remainder of this chapter, we focus on *Or*-predicates only, and leave the strategies for *And*-predicates to the reader. For the *Or*-predicate, if we know that one of the children's *val* attribute evaluates to *True*, we can commit the choice to that child. We express this as follows, using a function *choose<sub>or</sub>* (to be defined later):

```
sem Pred | Or -- Rules for production Or of nonterm Pred.
  left.env = lhs.env           -- Copies down env to the left.
  right.env = lhs.env          -- Copies down env to the right.
  merge left right as res = chooseor -- Creates child res using chooseor to merge.
  lhs.val = res.val           -- Pass val up from child res.
```

The merge-rule transforms children *left* and *right* into a single virtual child *res*. We may refer to the synthesized attributes of *res*, but not to those of *left* and *right*. Similarly, we need to define the inherited attributes of *left* and *right*, but may not define those of *res*.

The function *choose<sub>or</sub>* takes the synthesized attributes of *left* and *right* as arguments, and is required to provide values for the synthesized results for *res*<sup>3</sup>. As initial attempt, we define *choose<sub>or</sub>* as:

```
chooseor :: Bool → Bool → Bool
chooseor left_val right_val -- Synthesized attributes of left and right as parameter.
```

<sup>3</sup>Due to Haskell's laziness, on-demand evaluation starts as soon as we scrutinize a value of an attribute.

## 6.2 Example of a Stepwise AG for a Predicate Language

```
| left_val  = left_val  -- Takes the left value, or
| right_val = right_val -- takes right value, or
| otherwise = left_val  -- otherwise takes the left value.
```

With this function, we unambiguously specify what kind of solution we want. However, in terms of evaluation we are back where we started: we evaluate the entire left child first, thus evaluation is still left biased.

Scrutinizing the value of a synthesized attribute leaves us little control over the scheduling of the evaluation<sup>4</sup>. In the example, we cannot return a result until we make the choice, but in order to do so, we need to inspect the result, which causes one of the children to evaluate fully before we have a chance to inspect the other one. Instead, we want the scheduling decisions to be based on explicitly indicated intermediate results. On-demand evaluation does not help us here, and therefore we propose a different evaluation scheme to choose between alternatives.

This leads us to the second idea of this chapter: we evaluate a child under a choice strictly (instead of on-demand), thus computing attributes of children in a fixed order<sup>5</sup>. Instead of completely evaluating all synthesized attributes of such child, however, we evaluate it just far enough to yield a progress report. It then suspends to be resumed later. We explain later how to emit such progress reports during evaluation. To explore or merge two children both gradually and simultaneously, as well as report intermediate results to the parent, we take their progress reports alternatively, and intertwine them (explained below).

In our example, we wish to prioritize evaluation to the child that we estimate has performed the least amount of work, to balance out evaluation. For that, the progress reports need to give an indication that some work has been done. Hence, we define a type for progress reports (e.g. *Info*), with a plain constructor *Work* as possible value:

```
data Info = Work -- Application-specific type defined by programmer.
```

The value for a child that is passed to the *choose\_or* function is not just the value of the synthesized attribute, but now a *stepwise computation* of the type *Stepwise Info Bool*, where *Info* is the type of the progress reports, *Bool* is the type of the synthesized attribute. The type of *choose\_or* changes to:

$$\begin{aligned} \text{choose}_{or} &:: \text{Stepwise Info Bool} \rightarrow \text{Stepwise Info Bool} \\ &\rightarrow \text{Stepwise Info Bool} \end{aligned}$$

A computation of the type *Stepwise Info Bool* can be manipulated with a simple monadic API (Section 6.4):

```
emit      :: i → Stepwise i ()           -- Type i equals Info here.
smallStep :: Stepwise i α → Report i α   -- Evaluates to next report.
```

<sup>4</sup> We can use Haskell's lazy evaluation to return lazy approximations of the final values as result. This, however, complicates the AG severely, as rules must be manually lifted to operate on approximations.

<sup>5</sup> In this chapter, we take the order of appearance of children in a production as strict evaluation order. UUAGC actually supports Ordered Attribute Grammars (OAGs) [Kastens, 1980] that takes attribute dependencies into account, and ensures that all attributes are well-defined (non-cyclic). Using OAGs, the order of appearance is not important.

## 6 AGs with Stepwise Evaluation

```

data Report  $i \alpha = Done \alpha \mid Step i (Stepwise i \alpha)$  -- Result of smallStep.
return ::  $\alpha \rightarrow Stepwise i \alpha$ 
( $\gg$ ) ::  $Stepwise i \alpha \rightarrow Stepwise i \beta \rightarrow Stepwise i \alpha$ 
( $\gg\equiv$ ) ::  $Stepwise i \alpha \rightarrow (\alpha \rightarrow Stepwise i \alpha) \rightarrow Stepwise i \alpha$ 

```

Function *emit* yields a progress report, and *smallStep* evaluates the computation until it is *Done* (with attribute values) or yields a *Step* (with continuation). Computations are composable via monadic operators. The monadic sequence ( $m_1 \gg m_2$ ) performs  $m_1$ , throws away its result, then performs  $m_2$  and delivers  $m_2$ 's result. The monadic bind  $m \gg\equiv f$  performs  $f$  parameterized with the result of  $m$ .

Using the above API, we now redefine *choose<sub>or</sub>* as:

```

chooseor left right = choose' (smallStep left) (smallStep right) where
  choose' (Done v) _ = if v then left else right -- Choose.
  choose' _ (Done v) = if v then right else left -- Choose.
  choose' (Step Work left') (Step Work right') -- Both yielded a Work.
    = emit Work  $\gg$  chooseor left' right' -- Pass on the report.

```

Both children perform a step. When one of them evaluates to *Done*, we inspect its attribute and make a choice. By replacing the expression with either *left* or *right*, we eliminate the other choice. Otherwise, we emit a step that a bit of work has been done (for the current node), and retain the choice between the continuations of the children. This strategy effectively encodes a breadth-first exploration of the children.

We emit a *Work* progress report for each *Var* node. To inject such reports, we extend the *Var* production with a special built-in nonterminal *Progress*<sup>67</sup>. This nonterminal has a single inherited attribute named *info*, and no synthesized attributes. Since it has no synthesized attributes, a child of this nonterminal is never evaluated during on-demand evaluation. During strict evaluation, however, each child is evaluated. In that case, the implementation of *Progress* yields the progress report that it took as parameter<sup>8</sup>. This is exactly the behavior that we want. We are not interested in progress reports during on-demand evaluation, but the more we are interested in them during strict evaluation.

```

data Pred  $\mid Var \ report : Progress$  -- Additional child of production Var.
sem Pred  $\mid Var \ report.info = Work$  -- Defines its inherited attribute.

```

The *info* attribute is defined by a conventional rule, thus as right-hand side, we have access to any intermediate result (not needed for this example).

<sup>6</sup> Alternatively, the additional child can be specified as a higher-order attribute using Higher-Order AGs [Vogt et al., 1989] (supported by UUAG), which does not require us to change the production.

<sup>7</sup> Actually, the *Progress* nonterminal can be implemented using a merge-rule (Appendix A).

<sup>8</sup> Visits to children may be omitted if the synthesized attributes of that child are not used. This is undesirable when the child may emit a progress report. In Appendix A, we explain that progress reports themselves can be considered to be a hidden attribute. Hence, visits to children that emit progress reports can never be omitted during stepwise evaluation. Also, referential transparency is preserved.



## 6.2.5 Hybrid On-demand and Stepwise Evaluation

The attribute rules are oblivious towards stepwise evaluation. They are still pure functions between attribute values, which is important to reason with AGs. Access to progress reports is only possible in merge functions. Additionally, this allows stepwise and on-demand evaluation to coexist.

Stepwise evaluation is strict, therefore it cannot deal with attributes defined in a cyclic way. However, we only need stepwise evaluation while making a choice: once only one alternative is left, we can continue with on-demand evaluation. For that purpose, we provide a function:

*lazyEval* :: *Stepwise*  $i$   $\alpha \rightarrow \alpha$  -- Runs computation lazily, returns syn attributes.

It takes a partially reduced child as parameter, ignores progress reports, and returns the attributes ( $\alpha$  is instantiated to a product of the attribute types) on which we may perform conventional on-demand evaluation. Further details for *lazyEval* can be found in Section 6.4.

The global picture is that we start with *lazyEval* at the root of the AST. When *lazyEval* needs an attribute value of a merged child, it asks for the result of the merge function (e.g. *choose<sub>or</sub>*). Consequently, *choose<sub>or</sub>* invokes *smallStep* on its children, gradually reducing the candidate children, and ultimately returns one of the residual children (e.g. *left*). Then *lazyEval* proceeds with this child.

To implement these ideas, we arrive at the third idea of this chapter: The AG is compiled to a special form of coroutines (Section 6.3). A coroutine is a function that may yield an intermediate result and then suspends. Its caller receives that result, and can reinvoke the coroutine again to resume its execution. This gives us the behavior of *smallStep*. For *lazyEval*, we need some special behavior: in that case, a coroutine resumes and runs immediately towards its end. It constructs only the administration needed to perform the remaining computations in an on-demand fashion. We give an implementation for these special coroutines in Section 6.4.

## 6.3 SAG Translation

A SAG is a conservative extension of an AG that adds the **merge** rule. We sketched its static semantics in the previous section. In this section, we sketch its denotational semantics: we describe how to map a SAG to a monadic Haskell program (Translation scheme in Appendix B). The monad is defined in Section 6.4. The SAG translation is largely based on a conventional translation to Haskell, as implemented in UUAG [Saraiva and Swierstra, 1999].

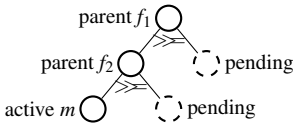
To translate a SAG, we translate each production to a coroutine (e.g. *sem.And*), implemented as a monad. With the coroutines we build an attributed tree (e.g. *taut* and *big<sub>1</sub>* in Section 6.2.1). This tree is represented as a *function* from the inherited attributes to a product of the synthesized attributes (and, as mentioned in the previous section, lifted in the stepwise monad). We call this tree the *semantic value* or simply the *semantics* (of the associated nonterminal/production). Thus, a coroutine is a function that takes the *semantic values* (or, simply: *semantics*) of its children as parameter, and returns its own semantics. For example, in case of nonterminal *Pred*, the type of its semantics (named *I.Pred*) and the signature of *sem.And* are:

## 6 AGs with Stepwise Evaluation

```
type  $I\_Pred = Env \rightarrow Stepwise\ Info\ Bool$  -- Function type for attributed tree.
sem $\_And :: I\_Pred \rightarrow I\_Pred \rightarrow I\_Pred$  -- Coroutine for And production.
```

We first discuss a translation of conventional AGs (*sem\_And* does not use **merge**). Coroutine *sem\_And* takes the semantics of its children as parameter. It must return its own semantics, which is a function from its single inherited attribute *env* to its single synthesized attribute *val* lifted in the stepwise monad. We use the encoding *childXattr* to unambiguously refer to an attribute, where *X* equals *I* for an inherited attribute and *S* for a synthesized attribute. The function body—the monad—comprises the *plan* for a production: it consists of the calls to the children, and definitions of the attributes. At a function call to a child, we pass its inherited attributes, then we obtain a monadic value with the synthesized attributes, which we can match against. Recursive do-notation [Erkök and Launchbury, 2000] allows us to write such plans concisely.

```
sem $\_And\ left\ right = \lambda lhsIenv \rightarrow \mathbf{do\ rec}$  -- Takes inh attr env.
   $leftSval \leftarrow left\ leftIenv$  -- Calls left child.
   $rightSval \leftarrow right\ rightIenv$  -- Calls right child.
  let  $leftIenv = lhsIenv$ 
  let  $rightIenv = lhsIenv$ 
  let  $lhsSval = leftSval \wedge rightSval$ 
  return  $lhsSval$ 
```



Under the hood, the do-notation reorders the statements to produce a linearized plan as a sequence of monadic binds<sup>9</sup>, e.g. (let-bindings omitted, replaced by dots):

$$left\ leftIenv \gg= (\lambda \dots \rightarrow right\ rightIenv \gg= (\lambda \dots \rightarrow return\ lhsSval))$$

A monadic bind  $m \gg= f$ , expresses that the remainder of the plan *f* is parametrized with the results of plan *m* (see the figure above). Strict evaluation goes from left to right, thus reducing a plan gradually.

To translate the merge rule, consider the translation for *sem\_Or*. The calls to the involved children are not made part of the plan: we do not match against their results. Instead, *choose\_or* takes the plans (stepwise computations) of its input children, and as result, must provide a plan (computation) for its output child. Subsequently, we match against the output child to obtain the synthesized attribute *res.val* of merged child *res*:

```
sem $\_Or\ left\ right = \lambda lhsIenv \rightarrow \mathbf{do\ rec}$  -- Function from lhs.env to lhs.val.
   $resSval \leftarrow choose\_or\ (left\ leftIenv)\ (right\ rightIenv)$  -- Translation for merge.
  let  $leftIenv = lhsIenv$  -- Defines env for left.
  let  $rightIenv = lhsIenv$  -- Defines env for right.
  let  $lhsSval = resSval$  -- Takes val attribute from res child.
  return  $lhsSval$  -- Returns result for Or production.
```

<sup>9</sup> In general, also calls to *mfix* are inserted to deal with cyclic definitions. We provide a definition for *mfix* in the library associated to this chapter. In practice, with UUAG, we use Ordered Attribute Grammars, that result in a more sophisticated translation that only needs non-recursive do-notation, without *mfix*.

SAGs are thus only a modest extension to the conventional translation. Most of the additional complexity is hidden in the implementation of the coroutine (Section 6.4).

## 6.4 Lazy Coroutines and the Stepwise Monad

We use a coroutine to represent the residual attributed tree after strict evaluations. We either transform this coroutine to its lazy result via *lazyEval*, or run it greedily using *smallStep* to to the point where it yields the next progress report. The data type *Stepwise i a* represents such a coroutine. It exposes the following structure:

```
data Stepwise i a where    -- Stepwise is a ‘defunctionalized’ monad.
  Yield  :: i → Stepwise i a → Stepwise i a  -- Paused computation
  Fail   :: String          → Stepwise i a    -- Aborted computation
  Return :: a                → Stepwise i a    -- Finished computation
  Pending :: Stepwise i b → Parents i b a → Stepwise i a

data Parents i a b where  -- Parent stack (root of type b, active child a).
  None   :: Parents i a a    -- Bottom of the stack.
  Bind   :: (a → Stepwise i z) → Parents i z b → Parents i a b
```

*Yield* means that the coroutine paused to yield a progress report of the type *i*. The second component represents the continuation. *Fail* represents an aborted computation, and *Return* means it succeeded, providing a value of type *a*.

During strict evaluation, reduction of a child starts only when its preceding sibling is fully reduced (linear execution of the plans in Section 6.3). Hence, every node has at most one child active, and a continuation of what to do after that child is finished. A *Pending* value encodes this: the first component is the deepest child awaiting further evaluation. The second component is the stack of parent-continuations. The GADT *Parents i a b* represents the parent nodes that await a value of type *a*, to ultimately compute a value of type *b*. When we match *None*, we reached the bottom of the stack, where we ensure that *b* equals *a*.

We can now give a *Monad* instance for *Stepwise*. The bind  $m \gg\equiv f$  is encoded as active child *m* with single pending parent *f*:

```
instance Monad (Stepwise i) where    -- Via the monad combinators, we
  return = Return                    -- thus build a Stepwise-value, and
  fail    = Fail                      -- reduce this value via
  m >>≡ f = Pending m (Bind f None)  -- smallStep or lazyEval.
  emit i = Yield i (return ())
```

Given a coroutine, we can run it immediately to its lazy result value. This process describes how we transform a residual tree into a tree on which we can perform on-demand evaluation. We step over any progress reports it may yield in the process. When we encounter a *Pending*, we apply *lazyEval* to get a lazy result of the child, and use *evalPending* to provide it to its parent, which in turn passes it to its parent, until we reach the root:

## 6 AGs with Stepwise Evaluation

```

lazyEval :: Stepwise i a → a           -- Interprets computation lazily.
lazyEval (Yield _ m) = lazyEval m      -- Skips progress report.
lazyEval (Fail s)   = error s         -- Interpreted as ⊥.
lazyEval (Return v) = v
lazyEval (Pending m p) = evalPending p (lazyEval m)
evalPending :: Parents i a b → a → b  -- lazyEval on chain of parents.
evalPending None    a = a             -- Reached the root.
evalPending (Bind f r) a = evalPending r (lazyEval (f a))

```

Given a coroutine, we can also run it greedily until it yields the next progress report. Either it fails, is finished, or is paused with the yielded information *i* and the continuation to resume it with:

```

data Report i a where           -- Outcome of smallStep.
  Failed :: String             → Report i a -- Aborted with message.
  Done   :: a                 → Report i a -- Finished with value.
  Step   :: i → Stepwise i a → Report i a -- Paused with user report.

```

The function *smallStep* performs a strict reduction. Once it encounters a *Yield* it can stop and return a *Step*:

```

smallStep :: Stepwise i a → Report i a  -- Evaluate until next report.
smallStep (Yield i m) = Step i m      -- Pause after a yield.
smallStep (Fail s)   = Failed s
smallStep (Return v) = Done v
smallStep (Pending m p) = reduce m p  -- Continues with m, and possibly p.

```

For a *Pending*, its result depends on the reduction of the active child. If it finishes without yielding a progress report, we pass the result to its parent and continue reducing the parent. If the active child itself turns out to be a *Pending*, we push its stack on the stack we already have, and continue reduction:

```

reduce :: Stepwise i a → Parents i a b → Report i b
reduce (Yield i m) r = Step i (Pending m r) -- Keeps residual parents.
reduce (Fail s) _ = Failed s
reduce m None = smallStep m -- No parents left.
reduce (Return v) (Bind f r) = reduce (f v) r -- Proceeds with parent f.
reduce (Pending m r') r = reduce m $push r' r -- Concatenates stacks.

push :: Parents i a b → Parents i b c → Parents i a c
push None r = r -- Appends to the bottom.
push (Bind f r') r = Bind f (push r' r) -- Walk towards the bottom.

```

The merging of the parent stacks is important. The stack represents a whole subtree, with the active child on top, and the root of the subtree on the bottom. When we want to reduce this subtree one step, we can thus immediately reduce the active child without having to traverse through the parents.

We provide the API that we discussed in this section as a Haskell library. Its implementation satisfies the monad laws (Appendix H) for both *lazyEval* and *smallStep* evaluation. Furthermore, if it holds that  $smallStep^* m = Done\ v$  then also *lazyEval* yields this value  $lazyEval\ m = v$ . The converse is not necessarily true:  $lazyEval\ (\perp \gg return\ ()) = ()$ , whereas  $smallStep^* (\perp \gg return\ ()) = \perp$ . However, when the AG is ordered, and each rule well-founded, then also the converse holds.

## 6.5 Imperative Implementation

The reference implementation that we presented in the previous section relies on several Haskell features, such as lazy evaluation, monads and GADTs. Despite that, the approach is portable to imperative languages and AG systems for these languages. As a proof of concept, we ported the example of Section 6.2 to Java (Appendix J), and implemented a small Stepwise support library<sup>10</sup>.

We encode demand-driven AGs as conventional for object oriented languages. AST nodes are represented by objects, using subclasses for productions. Attributes are encoded as fields on such nodes using getters and setters together with lazy initialization. We map each rule to a *Runnable* object. A rule is associated with one or more attributes. When the value of an attribute is not yet defined, the associated rule is executed, and the rule defines these attributes using side effect. A rule may refer to the values of other attributes, thus driving on-demand evaluation. Executing a rule twice has no effect.

Additionally, a node exposes a visit method that encodes the coroutine for strict evaluation. The visit method may be invoked multiple times, and returns either with progress information, or a pointer to a child node to evaluate first, or indicates that evaluation is done for the subtree. With each execution, the visit method executes some of the rules. Nodes can thus be evaluated strictly via the visit method, and on-demand by accessing the attributes directly.

As discussed in Section 6.3, children of a production come in two fashions: conventional children and merged children. Conventional children are conventional AST nodes, which expose inherited attributes. Merged children do not expose inherited attributes. To start evaluation and access the synthesized attributes from both types of children, a stepwise computation can be requested from a child. A stepwise computation is a coroutine-object that supports the *lazyEval* and *smallStep* operations of Section 6.4. It represents the evaluation of the subtree rooted by the child.

The stepwise computation obtained from a conventional child  $x$  represents the stack of nodes under strict evaluation. A node on the stack has been partially evaluated strictly and is waiting for strict evaluation of nodes above it to complete before proceeding with strict evaluation. The child  $x$  is on the bottom of the stack. The active child is the top of the stack. For the *lazyEval* operation, the stepwise computation obtained from a conventional child directly returns  $x$ . This corresponds to the *evalPending* operation, except that in contrast to the functional implementation, we do not have to thread the lazy outcomes around because these are represented by pointers and side effect in the imperative implementation. For *smallStep*, strict

<sup>10</sup> Stepwise Java example: <https://svn.science.uu.nl/repos/project.ruler.papers/archive/jstepwise.jar>

## 6 AGs with Stepwise Evaluation

evaluation proceeds with the active node. If it yields progress info, then these are returned as the report for *smallStep*, and the evaluation is effectively paused again. If evaluation of the active node is finished, the node is popped from the stack and evaluation continues with its parent. If an active node requests evaluation of a child first, these are pushed on the stack.

The stepwise computation obtained from a merged child is an object with a visit method that has access to the stepwise computation of the children that are being merged. With each visit, it must return a progress report. In particular, it can report that the computation is to be replaced with a computation of the children, thereby resolving the choice. Essentially, such a computation is an imperative version of the monadic *choose<sub>or</sub>*.

The imperative implementation closely resembles the functional implementation, although more verbosely. A language with native support for coroutines would simplify the implementation slightly.

## 6.6 Remarks

### 6.6.1 Extensions

We used SAGs to rapidly prototype a type-directed transformation. It required a small extension of the presented ideas: *semantic lookahead* (Appendix C). Often, a choice for a subtree has consequences at another location in tree. We thus (may) need to investigate potential alternatives beyond the evaluation of the subtree. To this end, we added a mechanism to obtain the continuation, and investigate the steps coming out of the continuation.

To combine progress reports of different types, and allow them to depend on the result type of the computation itself, we implemented *transcoding* (Appendix D). It can also be used to replace multiple reports by a single report (compression) to trade interleaving granularity with fewer reports to examine.

Finally, we offer *explicit sharing* via references. We use this mechanism to deal with the *MonadFix* instance required for recursive do-notation, and also to offer *memoization* (to turn the AG under user-defineable conditions into a graph).

### 6.6.2 Benchmarks

We benchmarked our approach (Appendix I) on a standard MacBook 2.1 with a 2 GHz dual-core processor, 2 GB of main memory, and GHC 6.12.1. We compared the execution time of code generated the conventional way by UUAG against code that uses stepwise evaluation, and measured the runtime overhead. The throughput of stepwise binds is about thirty times slower than the bind of the identity monad. The overhead is constant for *nextStep*, and is a bit more erratic for *lazyEval* (but comparable). On the other hand, the overhead is negligible in practice. We compared the compilation speeds of UHC, which makes heavy use of AGs (for both large and small tasks). The compilation time only marginally increased, and stays under random noise induced by garbage collection.

## 6.7 Related Work

This chapter is heavily inspired by uu-parsinglib [Swierstra, 2009, Hughes and Swierstra, 2003]. The parsing library supports both context-free and monadic grammars, and offers online results as well as error correction. It offers a data-type similar to our *Stepwise*. The essential difference is that uuparsing-lib's bind cannot yield a result until the LHS of the bind is fully step-wise evaluated<sup>11</sup>. Instead, required for the hybrid evaluation model, our implementation can yield a result when the RHS can do so (when using *lazyEval*). Also, uuparsing-lib's implementation manually manages stacks with semantic values computed so far (the outcome of the history parser), or semantic values still to come (the outcome of the future parser). Instead, in our implementation, these values are implicitly represented as local variables in closures.

Our work is related to various disambiguation strategies. Some approaches allow ambiguous ASTs and impose syntactic restrictions to resolve these, e.g. by conditionally rejecting certain productions based on the AST structure, or prioritizing some productions over others [van den Brand et al., 2002]. Other approaches generate a parse forest and filter later, potentially using semantic information [Bravenboer et al., 2005]. In contrast, our approach does not require all alternatives to be available a priori, and works for also in case of infinite trees and in combination with nonterminal attributes.

AGs are traditionally considered to select productions deterministically based on the syntax of a language. Conditional Attribute Grammars [Boyland, 1996], as supported by the FNC-2 system [Jourdan and Parigot, 1991] and our experimental Ruler system [Middelkoop et al., 2010a], allow multiple definitions for a productions guarded by conditions. These conditions need to be evaluated first, thus offer limited control over the exploration of alternatives.

There are AG evaluators that generate Prolog code [Walsteijn and Kuiper, 1986, Paakki, 1991]. Such an approach that depends on a logic language is unacceptable for us, as it does not mix well with our existing Haskell code. In contrast, our approach can be implemented in an arbitrary general purpose host language. The lazy evaluation our Haskell implementation relies on, actually just represents on-demand evaluation that other AG approaches provide. Moreover, as sketched by Section 6.2.5, our approach is compatible with circular and remote AGs [Magnusson and Hedin, 2007].

There are several different techniques to deal with declarative aspects in the specifications of programming languages. We classify declarative sources in increasing complexity:

- *Deterministic*. Both the production selection and the value of attributes are purely functional. A problem in this class trivially fits an AG. For example, auto-boxing and implicit coercions fall in this category.
- *k pass*. A priori unknown values (type variables) and derivations (deferred judgments) are replaced with place holders. Nodes in the AST are traversed at most  $k$  times. A

<sup>11</sup> Formally: we can write a conventional AG as a lazy applicative functor. Monads are more expressive, hence we require the following equality to hold, which is not the case for uu-parsinglib:

$$p \star q \equiv p \gg\gg \lambda f \rightarrow q \gg\gg a \rightarrow \text{return } (f \ a)$$

## 6 AGs with Stepwise Evaluation

decision about a place holder must be taken during one of these traversals. Typically, information about place holders is maintained in a substitution (concrete assignments to variables) or constraints (symbolic representation of a deferred judgment). For example, Hindley-Damas-Milner type inference (algorithm W) has  $k = 1$ . After one traversal, a type variable either got assigned a concrete type, or it is fixed by a skolem constant (and generalized later). Haskell 98 overloading resolving is an example of  $k = 2$ . In the first pass, type equalities are resolved and class-constraints are collected. In the second pass, the class-constraints are resolved. Such problems can be dealt with in AGs using additional attributes for substitutions and constraints.

- *$\omega$  pass.* Declarative aspects that are resolved by fixpoint iteration falls in this class. This includes the class of type and effect systems. Also, resolution of Haskell's overloading in combination with functional dependencies falls in this class. AGs with circular references can be used to encode such problems [Magnusson and Hedin, 2007, Jones, 1990], or AGs that can express iteration [Middelkoop et al., 2010a].
- *Exploration of alternatives.* In the previous classes, declarative aspects are resolved by sufficiently constraining a value, where the constraints are a pure function of the (attributed) AST. In the exploration class, declarative aspects are resolved by exploring assignments to place holders. This requires instantiations for place holders to be enumerable. Haskell's overloading combined with overlapping instances, and the construction of equality coercions for GADTs fall in this class. This chapter positions itself in this class.
- *Undecidable.* Inference for some declarative aspects is undecidable. For example, a polymorphic type can in general not be inferred for an argument of a recursive function.

Haskell offers several library approaches for backtracking, via folklore *Maybe* and list monads to more advanced monads [Hinze, 2000, Kiselyov et al., 2005, Fischer et al., 2009] that deal with nondeterminism and laziness. Alternatives are only explored for a value when this value is required. However, the order of appearance of alternatives affect memory retainment and how online the results are. See the discussion in Swierstra [2009, Section 4.1].

Coroutines were considered for many compilation tasks [Marlin, 1980]. Nowadays, they are mostly used to implement producer-consumer patterns. Kastens [1980] showed how to compile multi-visit AGs to coroutines. In this chapter we apply coroutines in a different fashion. We use coroutines to expose explicitly indicated parts of the internal state of an AG evaluation, in order to describe exploration strategies.

There are several approaches for monadic coroutines in Haskell. These implementations have their roots in the folklore CPS monad to pause, abort and merge computations. Kiselyov's Iteratees [Kiselyov, 2008] come close to our implementation. However, technical differences aside, there is a conceptual difference. Typical coroutine implementations allow invocations to take additional arguments. Since the result may depend on the value of such a parameter, a lazy result cannot be given until the last invocation. Therefore, the hybrid evaluation model that we presented cannot be implemented with such coroutines.



## 6.8 Conclusion

We presented SAGs, a powerful language extension to Attribute Grammars to cope with declarative aspects in the semantics of programming languages. We stated our requirements in Section 6.1, and showed how our approach meets these demands by example in Section 6.2, and sketched the implementation in Section 6.3 and Section 6.4. We implemented SAGs in the UUAG system.

The idea central to our approach is to encode alternatives as a choice between children of a production, and resolve this based on stepwise inspection of intermediate results of these children in the form of progress reports.

As future work, we intend to replace the overloading mechanism as currently implemented in UHC using the new AG features as presented in this paper. Also, a remaining question is if the stepwise monad is powerful enough to simplify the implementation of `uu-parsinglib`.

In Appendix A, Appendix E, and Appendix B we go into more detail of the AG part of the story. Appendix C and Appendix D show improvements of the stepwise monad. Appendix F gives an example that makes use of the additional improvements. Appendix G shows how various declarative aspects can be expressed in terms of each other. Proofs for some of our claims related to the monad laws can be found in Appendix H, and benchmark results in Appendix I. Finally, we show a translation to Java in Appendix J. The thesis contains appendices A-D. The extended edition contains the remaining appendices.

## 6.A Progress Reports and their Emission

As we mentioned in Section 6.2, we annotate productions with the built-in nonterminal *Progress* to yield progress reports. Actually, this built-in nonterminal is only a notational convenience: it is definable in terms of the merge-syntax that we presented before. In this section, we show the implementation.

The data type *Progress*, as mentioned in this chapter, can be implemented as follows:

```
data Progress | Progress
attr Progress inh info :: a
sem Progress | Progress
merge as res : Progress = emit lhs.info >> return ()
```

It has a single inherited attribute, and a single production. This production has a single child *res*, merged out of an empty set of children. To define the semantics of this child, we thus do not get any semantics of children as parameter. Since *Progress* does not have any synthesized attributes, defining a semantics for *res* is straightforward: we return the empty tuple `()`.

It is not immediately clear why this implementation would work: nonterminals without any synthesized attributes never need to be visited. Referential transparency tells us that we may replace a child with a product of its synthesized attributes, and attribute references with the corresponding values. A child without synthesized attributes may never be evaluated, and the progress report never yielded.

## 6 AGs with Stepwise Evaluation

The essential realization here is that there is that a hidden attribute plays a role: the progress reports themselves are a purely functional attribute. Hence, during strict evaluation via *smallStep*, we still visit children without any explicitly declared synthesized attributes in order to get the progress reports. In contrast, during on-demand evaluation via *lazyEval*, we ignore progress reports, hence do not evaluate children without synthesized attributes.

### 6.B Translation Scheme

In this section, we formalize SAGs. We first define a small core language *sagcore*, consisting of Haskell extended with embedded AG blocks, obtained by desugering AG descriptions. The following grammar lists the syntax of these embedded AG blocks:

```
i ::= attr I inh  $\overline{a1}$  syn  $\overline{a2}$  -- attribute delcs
a ::= x :: hty -- attribute decl, with Haskell type hty
s ::= sem I  $\bar{r}$  -- semantics expr, defines production for I
r ::= p = e -- binds to p to pure e
      | child c :: I = e -- declares child
      | merge  $\bar{c}$  as c :: I = e -- declares merged child
o ::= x.x -- expression, attribute occurrence
x, I, p, e -- identifiers, patterns, expressions respectively
```

Attribute declarations *i* declares all attributes (name and type) of a nonterminal *I*. A semantics block defines a single production for a nonterminal *I*, and gives its rules  $\bar{r}$ . Productions in *sagcore* are nameless: we use a Haskell declaration to give it a name. Furthermore, we declare its children through rules. Rules either define attributes, or declare children: we introduce all children as higher-order attributes (Appendix E). The expressions *e* are Haskell expressions, with possible attribute occurrences *o*. Patterns *p* are Haskell patterns, also with possible attribute occurrences *o*.

For example, we show how the production *Or* as mentioned in Section 6.2 is encoded in *sagcore*. In UUAG-notation, we declare a production *Or*, declare the attributes of the corresponding nonterminal, and give the rules for the attributes:

```
data Pred | Or left, right :: Pred
attr Pred inh e :: Env syn b :: Bool
sem Pred | Or
  left.e = lhs.e
  right.e = lhs.e
  merge left right as res :: Pred = choose_or
  lhs.b = res.b
```

In *sagcore*, we declare the attributes of the nonterminal, then use a Haskell function to represent the production: it takes the semantics of the children as parameter, then uses an embedded semantics block to define the semantics for the production itself:

```

attr Pred inh e :: Env syn b :: Bool
sem_Pred_And l r =
  sem Pred
    child left :: Pred = l
    child right :: Pred = r
    merge left right as res :: Pred = choose_or
    left.e = lhs.e
    right.e = lhs.e
    lhs.b = res.b

```

We thus keep the rules, yet express the grammar directly as Haskell functions:

$$\begin{array}{l} \mathbf{data} \ N \mid C \bar{c} :: I \quad \rightsquigarrow \quad \mathbf{sem\_N\_C} \bar{c} = \mathbf{sem} \ N \\ \mathbf{sem} \ N \mid C \bar{r} \quad \quad \quad \mathbf{child} \ c :: I = c; \bar{r} \end{array}$$

With such a function that represents a production, we can construct attributed trees. Each node in the tree has its own set of inherited and synthesized attributes: the associated nonterminal specifies their name and their types. The rules of the production define the attributes of the production, and declare what attributes the children have. The production must define its synthesized attributes, and the inherited attributes of its children exactly once (with a correct type). Attribute references in the expressions may refer to the inherited attributes of the production, or the synthesized attributes of the children. There is one exception: the inherited attributes of the merged child (e.g. *res*) may not be defined, and the synthesized attributes of the merging children (e.g. *left* and *right*) may not be referred to.

We define a translation to Haskell (denotational semantics) that gives both a static and operational semantics to SAGs. If the generated Haskell program is type correct then so is the *sagcore* program. The execution of the generated Haskell functions shows how the rules are used to construct the attributed trees.

As mentioned in Section 6.3, we translate a semantics-block to an execution plan of the production. We use a naming conventional to translate AG names to Haskell names. Attributes are referred to by an identifier *cXa*. In this notation, *c* and *a* are the name of the child and the name of the attribute respectively. *X* is a subscript *I* for an inherited attribute, and *S* for a synthesized attribute. Merging children are prefixed with an underscore (we assume that names of children do not start with an underscore).

Each rule corresponds to an instruction in the execution plan. In the end, we return a tuple with values for the synthesized attributes:

$$\begin{array}{ll} \llbracket \mathbf{sem} \ I \ \bar{r} \rrbracket & \rightsquigarrow \lambda lhs_1 ins_1 \dots lhs_n ins_n \rightarrow \mathbf{do} \ \mathbf{rec} \\ & \quad \{ \llbracket \bar{r} \rrbracket; \mathbf{return} \ (lhs_{sout_1}, \dots, lhs_{sout_m}) \} \\ \llbracket \mathbf{child} \ c :: I = e \rrbracket, \quad \text{conventional} & \rightsquigarrow (c_{sout_1}, \dots, c_{sout_m}) \leftarrow e \ c_1 ins_1 \dots c_n ins_n \\ \llbracket \mathbf{child} \ c :: I = e \rrbracket, \quad \text{merged} & \rightsquigarrow \mathbf{let} \ \_c = e \ c_1 ins_1 \dots c_n ins_n \\ \llbracket p = e \rrbracket & \rightsquigarrow \mathbf{let} \ \llbracket p \rrbracket = \llbracket e \rrbracket \\ \llbracket \mathbf{merge} \ cs \ \mathbf{as} \ c : N = e \rrbracket & \rightsquigarrow (c_{sout_1}, \dots, c_{sout_m}) \leftarrow \llbracket e \rrbracket \ \_c_1 \dots \_c_k \end{array}$$

The indices *m* and *n* range over the inherited and synthesized attributes of nonterminal *I*.

In previous work [Middelkoop et al., 2010a], we described different (more sophisticated) translations of (Ordered) Attribute Grammars to execution plans (in Haskell). The merge-syntax is fully compatible with those translations.

## 6.C Semantic Lookahead

In this section, we show how to deal with a choice that has a potential global effect on attributes of the tree. For example, suppose that we deal with a type inferencer that needs to choose between *int* and *double* for the type of a numerical constant. This choice may have a global effect: a wrong choice potentially causes a typing error in the remainder of the program to type. To explore such a choice, we want to look at the steps of a child *and* the steps that the remaining computation gives if we would choose that child. In terms of the monad: if  $k$  is the continuation after the choice, i.e.  $\text{choose } l r \ggg k$ , then we want to lift the choice to  $\text{choose } (l \ggg k) (r \ggg k)$ .

We provide a monadic operation *Ahead* (explained below) to access the continuation  $k$ . This operations comes at a price: since the continuation is not known until runtime, so we wish our choose-function to work for arbitrary continuations. In particular, we refrain from making static assumptions on the type of the result the continuation computes.

**data** *Stepwise i a where*

*Ahead* :: (forall  $b$ . ( $a \rightarrow \text{Stepwise } i b$ )  $\rightarrow \text{Stepwise } i b$ )  $\rightarrow \text{Stepwise } i a$

*Ahead* takes as argument a function  $f$  that takes the continuation  $k$  as argument. To use *Ahead*, we provide this function  $f$ . The continuation takes the result that we are supposed to compute, and returns a computation of some existential type  $b$ . The computation passed to *ahead* thus wraps around the computation: it specifies an input for the continuation, and allows us to modify the result of the continuation. Thus, this lifted the choice to toplevel, as mentioned in the previous paragraph. Also, modifying the result of the continuation instead of building an input for the continuation based on trying out the continuation is what makes this approach different from Continuation Passing Style's *call/cc*.

As an example, a computation  $m$  is equivalent to *Ahead* ( $\lambda k \rightarrow m \ggg k$ ). As another example, a global choice can be made using:

*Ahead* ( $\lambda k \rightarrow \text{choose } (l \ggg k) (r \ggg k)$ )

Although we cannot make an assumption about the type of the result of  $k$ , we can make an assumption on the type of progress reports, and thus use the contents of the progress reports to direct the exploration between the two choices (depending on the search strategy).

When we encounter an *Ahead*  $f$  in *lazyEval*, the question is what continuation we pass in. The function  $f$  is required to get the computation that resembles the remaining continuation. However, since we are in *lazyEval*, the continuation cannot return any progress reports, and cannot observably fail. Thus we simply pass in *Return*, which succeeds immediately, and gives us the result that  $f$  passes to its continuation:

*lazyEval* (*Ahead*  $f$ ) =  $f$  *Return*

When we encounter an *Ahead f* in *smallStep* without parents on the parent stack, we pause the computation. The evaluation can only continue if the caller specifies how the computation proceeds (possibly by calling *Ahead* itself):

**data Report i a where**

$$\begin{aligned} \text{Lookahead} &:: (\text{forall } b.(a \rightarrow \text{Stepwise } i b) \rightarrow \text{Stepwise } i b) \rightarrow \text{Report } i a \\ \text{smallStep } (\text{Ahead } f) &= \text{Lookahead } f \end{aligned}$$

If there are parents on the parent stack, however, we continue to reduce *f*. The continuation to pass to *f* are the remaining parents on the parent stack:

$$\text{reduce } (\text{Ahead } f) (\text{Bind } g r) = \text{smallStep } \$f \$\lambda a \rightarrow \text{Pending } (g a) r$$

The use of *Ahead* has an interesting interplay with the hybrid evaluation model. On-demand evaluation skips progress reports, and passes a *Return* as continuation. If a parent of a child that uses lookahead is evaluated on-demand, then the lookahead of the child does not observe the skipped progress reports. So, lookahead does not see beyond on-demand evaluated AST nodes. So, if a progress report contains information essential to a choice using lookahead, we need to take sufficient *smallSteps* at a common ancestor node such that the lookahead observed the report. For example, we can emit a progress report e.g. *DoneGreedy* when a choice using lookahead inspected the progress reports it was interested in, take *smallSteps* at the root of the tree (or a common ancestor), until we encounter *DoneGreedy*, then switch to *lazyEval*.

Also, *Ahead* has an interplay with multi-visit AGs (derivable from Ordered Attribute Grammars). Without *Ahead*, stepwise evaluation yields a *Done* for a child when the *first* visit is finished. When using *Ahead*, however, yields a *Done* when the continuation finished with its *last* visit. Again, the progress report mechanism can be used to limit the exploration to certain visits, or yield the outcome of a visit as intermediate result.

It is advisable to ensure that all choices can be made based on results of the first visit. If it requires a progress report that is emitted in a second visit, this requires the first visit to finish completely, which is likely already a full exploration of the tree (for that choice). It is possible to make the approach more flexible and offer just on-demand evaluation for first visits, and (hybrid) stepwise evaluation for later visits. That would allow exploration in a later visit based on (lazily) computed results in earlier visits. This requires visits to be made explicit in the AG specification (Chapter 3 and Chapter 5).

## 6.D Watchers

With the approach described so far, the type *i* of a progress is fixed: given an  $m \ggg k$ , both *m* and the computation returned by *k* must have the same *i* type. This limitation affects the compositionality of stepwise computations. Also, sometimes we wish to return progress reports of the same type as the result of the computation<sup>12</sup>. On the other hand, since the type

<sup>12</sup> See Example 7 of `Examples.hs` in the cabal package at <https://svn.science.uu.nl/repos/project.ruler.papers/archive/stepwise-1.0.2.tar.gz>

## 6 AGs with Stepwise Evaluation

$i$  is fixed, we know that a continuation has this type, thus when using *Ahead* (Appendix C) we can inspect the progress reports of the continuation. This we cannot do without knowing  $i$ .

To get the best of both ways, we parametrize the type  $i$  with a type  $w$  that functions as an index, the *watcher*. A computation has the type *Stepwise i w a*, which returns progress reports with values of the type  $i w$ . When  $w$  is an existential type, e.g. when using *Ahead*, we can still scrutinize on all values not dependent on  $w$ . If  $w$  is a concrete type, we can scrutinize the depending values as well.

To embed a computation with a different watcher type, we provide a transcoding operation:

**data** *Stepwise i w a where*

*Transcode* :: ( $i v \rightarrow [i w]$ )  $\rightarrow$  *Stepwise i v a*  $\rightarrow$  *Stepwise i w a*

It takes a progress report of type  $i v$  and converts it to zero or more progress reports of type  $i w$ . For example, *smallStep (Transcode (const []) m)* does not return any progress reports. In the actual implementation, we maintain composed transcoders on the parent stack, such that we can immediately apply them without having to traverse the stack.

In practice, we also allow the transcoding function to store a local state, such that it can remember a number of progress reports and combine them into a single progress report (compression). If paths in the tree are long, and many nodes are inspecting and passing on progress reports, then each node gets many reports to process (especially near the root). With the transcoding mechanism, we can trade evaluation granularity for the number of reports.

## 7 AGs with Dependent Types

Attribute Grammars (AGs) are a domain-specific language for functional and composable descriptions of tree traversals. Given such a description, it is not immediately clear how to state and prove properties of AGs formally. To meet this challenge, we apply dependent types to AGs. In a dependently typed AG, the type of an attribute may refer to values of attributes. The type of an attribute is an invariant, the value of an attribute a proof for that invariant. Additionally, when an AG is cycle-free, the composition of the attributes is logically consistent. We present a lightweight approach using a preprocessor in combination with the dependently typed language Agda.

### 7.1 Introduction

Functional programming languages are known to be convenient languages for implementing a compiler [Appel, 1998]. As part of the compilation process, a compiler computes properties of Abstract Syntax Trees (ASTs), such as environments, types, error messages, and code. In functional programming, these syntax-directed computations are typically written as *catamorphisms*<sup>1</sup>. An *algebra* defines an inductive property in terms of each constructor of the AST, and a catamorphism applies the algebra to the AST. Catamorphisms thus play an important role in a functional implementation of a compiler.

Attribute Grammars (AGs) [Knuth, 1968] are a domain-specific language for *composable* descriptions of catamorphisms. AGs facilitate the description of complex catamorphisms that typically occur in complex compiler implementations.

An AG extends a context-free grammar by associating *attributes* with nonterminals. Functional *rules* are associated with productions, and define values for the attributes that occur in the nonterminals of associated productions. As AGs are typically embedded in a host language, the rules are terms in the host language, which may additionally refer to attributes. Attributes can easily be composed to form more complex properties. An AG can be compiled to an efficient functional algorithm that computes the synthesized attributes of the root of the AST, given the root's inherited attributes.

It is not immediately clear how to formally specify and write proofs about programs implemented with AGs. For example, it is common to prove that a type inferencer is a sound and complete implementation of a type system, and that the meaning of a well typed source program is preserved. *Dependent types* [Bove and Dybjer, 2009] provide a means to use

---

<sup>1</sup> Catamorphisms are a generalization of folds to tree-like data structures. We consider catamorphisms from the perspective of algebraic data types in functional programming instead of the equivalent notion in terms of functors in category theory. A catamorphism  $\text{cata}_\tau (f_1, \dots, f_n)$  replaces each occurrence of a constructor  $c_i$  of  $\tau$  in a data structure with  $f_i$ . The product  $(f_1, \dots, f_n)$  is called an *algebra*. An element  $f_i$  of the algebra is called a *semantic function*.

## 7 AGs with Dependent Types

*types* to encode properties with the expressiveness of (higher-order) intuitionistic propositional logic, and *terms* to encode proofs. Such programs are called correct by construction, because the program itself is a proof of its invariants. The goal of this chapter is therefore to apply dependent types to AGs, in order to formally reason with AGs.

Vice versa, AGs also offer benefits to dependently typed programming. Because of the Curry-Howard correspondence, dependently typed AGs are a domain-specific language to write structurally inductive proofs in a *composable, aspect-oriented* fashion; each attribute represents a separate aspect of the proof. Additionally, AGs alleviate the programmer from the tedious orchestration of multi-pass traversals over data structures, and ensure that the traversals are *total*: totality is required for dependently typed programs for reasons of logical consistency and termination of type checking. Hence, the combination of dependent types and AGs is mutually beneficial.

We make the following contributions in this chapter:

- We present the language  $AG_{DA}$  (Section 7.3), a light-weight approach to facilitate dependent types in AGs, and vice versa, AGs in the dependently typed language Agda.  $AG_{DA}$  is an embedding in Agda via a preprocessor.

In contrast to conventional AGs, we can encode invariants in terms of dependently typed attributes, and proofs as values for attributes. This expressiveness comes at a price: to be able to compile to a total Agda program, we restrict ourselves to the class of ordered AGs, and demand the definitions of attributes to be total.

- We define a desugared version of  $AG_{DA}$  programs (Section 7.4) and show how to translate them to plain Agda programs (Section 7.5).
- Our approach supports a conditional attribution of nonterminals, so that we can give total definitions of what would otherwise be partially defined attributes (Section 7.6).

In Section 7.2, we introduce the notation used in this chapter. However, we assume that the reader is both familiar with AGs (see [Löh et al., 1998]) and dependently typed programming in Agda (see [Norell, 2009]).

## 7.2 Preliminaries

In this warm-up section, we briefly touch upon the Agda and AG notation used throughout this chapter. As an example, we implement the sum of a list of numbers with a catamorphism. We give two implementations: first one that uses plain Agda, then another using  $AG_{DA}$ . This example does not yet use dependently typed attributes. These are introduced in the next section.

In the following code snippet, the data type *List* represents a cons-list of natural numbers. The type  $T'List$  is the type of the value we compute (a number), and  $A'List$  is the type of an algebra for *List*. Such an algebra contains a *semantic function* for each constructor of *List*, which transforms a value of that constructor into the desired value (of type  $T'List$ ), assuming that the transformation has been recursively applied to the fields of the constructor. The catamorphism  $cata_{List}$  performs the recursive application.



```

data List : Set where           -- represents a cons-list of natural numbers
  nil  : List                    -- constructor has no fields
  cons : ℕ → List → List        -- constructor has a number and tail list as fields

T'List = ℕ                       -- defines a type alias T'List : Set,
A'List = (T'List, ℕ → T'List → T'List) -- and A'List : Set

cataList : A'List → List → T'List -- applies algebra to list
cataList (n, _) nil = n             -- in case of nil, replaces nil with n
cataList alg l with alg | l        -- otherwise, matches on alg and l
cataList alg l | (-, c) | cons x xs with cataList alg xs -- recurses on xs
cataList alg l | (-, c) | cons x xs | r = c x r          -- replaces cons with c

```

In Agda, a function is defined by one or more equations. A with-construct facilitates pattern matching against intermediate values. An equation that ends with with  $e_1 \mid \dots \mid e_n$  parameterizes the equations that follow with the values of  $e_1, \dots, e_n$  as additional arguments. Vertical bars separate the patterns intended for the additional parameters.

The actual algebra itself simply takes 0 for the *nil* constructor, and  $_{-}+_{-}$  for the *cons* constructor. The function  $sum_{List}$  shows how the algebra and catamorphism can be used.

```

semnil : T'List                -- semantic function for nil constructor
semnil = 0                      -- T'List = ℕ (defined above)

semcons : ℕ → T'List → T'List -- semantic function for cons constructor
semcons = _+_                  -- _+_ : ℕ → ℕ → ℕ (defined in library)

sumList : List → T'List         -- transforms the List into the desired sum
sumList = cataList (semnil, semcons) -- algebra is semantic functions in a tuple

```

In the example, the sum is defined in a bottom-up fashion. By taking a function type for  $T'List$ , values can also be passed top-down. Multiple types can be combined by using products. Such algebras quickly become tedious to write. Fortunately, we can use AGs as a domain-specific language for algebras. In the code below, we give an AG implementation: we specify a grammar that describes the structure of the AST, declare attributes on productions, and give rules that define attributes.

We now give an implementation of the same example using  $AG_{DA}$ . The code consists of blocks of plain Agda code, and blocks of AG code. To ease the distinction, Agda's keywords are underlined, and keywords of  $AG_{DA}$  are typeset in bold.

A grammar specification is a restricted form of a data declaration (for an AST): data constructors are called *productions* and their fields are explicitly marked as *terminal* or *nonterminal*. A nonterminal field represents a *child* in the AST and has attributes, whereas a terminal field only has a value. A plain Agda data-type declaration can be derived from a grammar specification. In such a specification, nonterminal types must have a fully saturated, outermost type constructor that is explicitly introduced by a grammar declaration. Terminal types may be arbitrary Agda types<sup>2</sup>.

<sup>2</sup> In general, although not needed in this example, nonterminal types may be parametrized, production types may refer to its field names, and field types may refer to preceding field names.

## 7 AGs with Dependent Types

```

grammar List : Set -- declares nonterminal List of type Set
prod nil : List -- production nil of type List (no fields)
prod cons : List -- production cons of type List (two fields)
term hd : ℕ -- terminal field hd of type ℕ
nonterm tl : List -- nonterminal field tl of type List

```

With an interface specification, we declare attributes for nonterminals. Attributes come in two fashions: *inherited* attributes (used in a later example) must be defined by rules of the parent, and *synthesized* attributes may be used by the parent. Names of inherited attributes are distinct from names of synthesized attributes; an attribute of the same name and fashion may only be declared once per nonterminal. We also partition the attributes in one or more *visits*. These visits impose a partial order on attributes. Inherited attributes may not be defined in terms of a synthesized attributes of the same visit or later. We use this order in Section 7.4 to derive semantic functions that are total.

```

itf List -- interface for nonterminal List,
visit compute -- with a single visit that is named compute,
syn sum : ℕ -- and a synthesized attribute named sum of type ℕ

```

Finally, we define each of the production's attributes. We may refer to an attribute using *child.attr* notation. For each production, we give rules that define the inherited attributes of the children and synthesized attributes of the production itself (with *lhs* as special name), using inherited attributes of the production and synthesized attributes of the children. The special name *loc* refers to the terminals, and to local attributes that we may associate with a production.

```

datasem List -- defines attributes of List for constructors of List
prod nil lhs.sum = 0 -- rule for sum of production nil
prod cons lhs.sum = loc.hd + tl.sum -- refers to terminal hd and attr tl.sum

```

The left-hand side of a rule is a plain Agda pattern, and the right-hand side is either a plain Agda expression or with-construct (not shown in this example). Additionally, both the left and right-hand sides may contain attribute references.

During attribute evaluation, visits are performed on children to obtain their associated synthesized attributes. We do not have to explicitly specify when to visit these children, neither is the order of appearance of rules relevant. However, an inherited attribute *c.x* may not depend on a synthesized attribute *c.y* of the same visit or later (in the interface). This guarantees that the attribute dependencies are acyclic, so that we can derive when children need to be visited and in what order.

AGs are a domain-specific language to write algebras in terms of attributes. From the grammar, we generate the data type and catamorphism. From the interface, we generate the  $T'List$  type. From the rules, we generate the semantic functions  $sem_{nil}$  and  $sem_{cons}$ . AGs pay off when an algebra has many inherited and synthesized attributes. Also, there are many AG extensions that offer abstractions over common usage patterns (not covered in this chapter). In the next section we present AGs with dependent types, so that we can formulate properties of attributes (and their proofs).

## 7.3 Dependently Typed Example

In this section, we use  $\text{AG}_{\text{DA}}$  to implement a mini-compiler that performs name checking of a simple language *Source*, and translates it to target language *Target* if all used identifiers are declared, or produces errors otherwise. A term in *Source* is a sequence of identifier definitions and identifier uses, for example:  $\text{def } a \diamond \text{use } b \diamond \text{use } a$ . In this case,  $b$  is not defined, thus the mini-compiler reports an error. Otherwise, it generates a *Target* term, which is a clone of the *Source* term that additionally carries evidence that the term is free of naming errors. Section 7.3.2 shows the definition of both *Source* and *Target*.

We show how to prove that the mini-compiler produces only correctly named *Target* terms and errors messages that only mention undeclared identifiers. The proofs are part of the implementation's code. Name checking is only a minor task in a compiler. However, the example shows many aspects of a more realistic compiler.

### 7.3.1 Support Code Dealing With Environments

We need some Agda support code to deal with environments. We show the relevant data structures and type signatures for operations on them, but omit the actual implementation. See Section 7.A for more details about the actual implementation. We represent the environment as a cons-list of identifiers.

$$\begin{aligned} \text{Ident} &= \text{String} && \text{-- Ident : Set} \\ \text{Env} &= \text{List Ident} && \text{-- Env : Set} \end{aligned}$$

In intuitionistic type theory, a data type represents a relation, its data constructors deduction rules for such a relation, and values built using these constructors are proofs for instances of the relation. We use some data types to reason with environments.

A value of type  $\iota \in \Gamma$  is a proof that an identifier  $\iota$  is member of an environment  $\Gamma$ . A value *here* indicates that identifier is at the front of the environment. A value *next* means that the identifier can be found in the tail of the environment, as described by the remainder of the proof.

$$\begin{aligned} \underline{\text{data}} \text{ } \_ \in \_ : & \text{Ident} \rightarrow \text{Env} \rightarrow \text{Set} \text{ where} \\ \text{here} : & \{ \iota : \text{Ident} \} \{ \Gamma : \text{Env} \} \rightarrow \iota \in (\iota :: \Gamma) \\ \text{next} : & \{ \iota_1 : \text{Ident} \} \{ \iota_2 : \text{Ident} \} \{ \Gamma : \text{Env} \} \rightarrow \iota_1 \in \Gamma \rightarrow \iota_1 \in (\iota_2 :: \Gamma) \end{aligned}$$

The type  $\Gamma_1 \sqsubseteq \Gamma_2$  represents a proof that an environment  $\Gamma_1$  is contained as a substring (with each mapping as a symbol) of an environment  $\Gamma_2$ . A value *subLeft* means that the environment  $\Gamma_1$  is a prefix of  $\Gamma_2$ , and *subRight* means that  $\Gamma_1$  is a suffix. With *trans*, we transitively compose two proofs.

$$\begin{aligned} \underline{\text{data}} \text{ } \_ \sqsubseteq \_ : & \text{Env} \rightarrow \text{Env} \rightarrow \text{Set} \text{ where} \\ \text{subLeft} : & \{ \Gamma_1 : \text{Env} \} \{ \Gamma_2 : \text{Env} \} \rightarrow \Gamma_1 \sqsubseteq (\Gamma_1 ++ \Gamma_2) \\ \text{subRight} : & \{ \Gamma_1 : \text{Env} \} \{ \Gamma_2 : \text{Env} \} \rightarrow \Gamma_2 \sqsubseteq (\Gamma_1 ++ \Gamma_2) \\ \text{trans} : & \{ \Gamma_1 : \text{Env} \} \{ \Gamma_2 : \text{Env} \} \{ \Gamma_3 : \text{Env} \} \rightarrow \Gamma_1 \sqsubseteq \Gamma_2 \rightarrow \Gamma_2 \sqsubseteq \Gamma_3 \rightarrow \Gamma_1 \sqsubseteq \Gamma_3 \end{aligned}$$

## 7 AGs with Dependent Types

The following functions operate on proofs. When an identifier occurs in an environment, function *inSubset* produces a proof that the identifier is also in the superset of the environment. Given an identifier and an environment,  $\iota \in? \Gamma$  returns either a proof  $\iota \in \Gamma$  that the element is in the environment, or a proof that it is not.

$$\begin{aligned} \text{inSubset} &: \{ \iota : \text{Ident} \} \{ \Gamma_1 : \text{Env} \} \{ \Gamma_2 : \text{Env} \} \rightarrow \Gamma_1 \sqsubseteq \Gamma_2 \rightarrow \iota \in \Gamma_1 \rightarrow \iota \in \Gamma_2 \\ \_ \in? \_ &: (\iota : \text{Ident}) \rightarrow (\Gamma : \text{Env}) \rightarrow \neg(\iota \in \Gamma) \uplus (\iota \in \Gamma) \end{aligned}$$

A value of the sum-type  $\alpha \uplus \beta$  either consists of an  $\alpha$  wrapped in a constructor *inj<sub>1</sub>* or of a  $\beta$  wrapped in *inj<sub>2</sub>*.

### 7.3.2 Grammar of the Source and Target Language

Below, we give a grammar for both the *Source* and *Target* language, such that we can analyze their ASTs with AGs<sup>3</sup>. The *Target* language is a clone of the *Source* language, except that terms that have identifiers carry a field *proof* that is evidence that the identifiers are properly introduced.

```

grammar Root      : Set          -- start symbol of grammar and root of AST
prod root : Root  nonterm top : Source -- top of the Source tree

grammar Source    : Set          -- grammar for nonterminal Source
prod use         : Source        -- 'result type' of production
term             $\iota$  : Ident      -- terminals may have arbitrary Agda types
prod def         : Source        -- 'result type' may be parametrized
term             $\iota$  : Ident
prod _ $\diamond$ _      : Source        -- represents sequencing of two Source terms
nonterm left    : Source        -- nonterminal fields must have a nonterm as
nonterm right   : Source        -- outermost type constructor.

grammar Target    : Env  $\rightarrow$  Set -- grammar for nonterminal Target
prod def          : Target  $\Gamma$     -- production type may refer to any field,
term?            $\Gamma$  : Env        -- e.g.  $\Gamma$ . Agda feature: implicit terminal
term             $\iota$  : Ident      -- (inferred when building a def)
term             $\phi$  :  $\iota \in \Gamma$  -- field type may refer to preceding fields
prod use         : Target  $\Gamma$ 
term?            $\Gamma$  : Env        -- a Target term carries evidence: a
term             $\iota$  : Ident      -- proof that the identifier is in the
term             $\phi$  :  $\iota \in \Gamma$  -- environment
_ $\diamond$ _          : Target  $\Gamma$ 
term?            $\Gamma$  : Env
nonterm left     : Target  $\Gamma$     -- nonterm fields introduce children that
nonterm right    : Target  $\Gamma$     -- have attributes

data Err : Env  $\rightarrow$  Set where    -- data type for errors in Agda notation

```

<sup>3</sup> In our example, we could have defined the type *Target* using conventional Agda notation instead. However, the grammar for *Target* serves as an example of a parameterized nonterminal.

$$\begin{aligned} \text{scope} &: \{\Gamma : \text{Env}\} (\iota : \text{Ident}) \rightarrow \neg(\iota \in \Gamma) \rightarrow \text{Err } \Gamma \\ \text{Errs } \Gamma &= \text{List } (\text{Err } \Gamma) \quad \text{-- Errs : Env} \rightarrow \text{Set} \end{aligned}$$

As shown in Section 7.2, we generate Agda data-type definitions and catamorphisms from this specification.

The concrete syntax of the source language *Source* and target language *Target* of the mini-compiler is out of scope for this chapter; the grammar defines only the abstract syntax. Similarly, we omit a formal operational semantics for *Source* and *Target*: it evaluates to unit if there is an equally named *def* for every *use*, otherwise evaluation diverges.

### 7.3.3 Dependent Attributes

In this section, we define *dependently typed* attributes for *Source*. Such a type may contain references to preceding<sup>4</sup> attributes using *inh.attrNm* or *syn.attrNm* notation, which explicitly distinguishes between inherited and synthesized attributes. The type specifies a property of the attributes it references; an attribute with such a type represents a proof of this property.

In our mini-compiler, we compute bottom-up a synthesized attribute *gathEnv* that contains identifiers defined by the *Source* term. At the root, the *gathEnv* attribute contains all the defined identifiers. We output its value as the synthesized attribute *finEnv* (final environment) at the root. Also, we pass its value top-down as the inherited attribute *finEnv*, such that we can refer to this environment deeper down the AST. We also pass down an attribute *gathInFin* that represents a proof that the final environment is a superset of the gathered environment. When we know that an identifier is in the gathered environment, we can thus also find it in the final environment. We pass up the attribute *outcome*, which consists either of errors, or of a correct *Target* term.

```

itf Root      -- attributes for the root of the AST
  visit compile syn finEnv  : Env
                syn outcome : (Errs syn.finEnv)  $\uplus$  (Target syn.finEnv)

itf Source    -- attributes for Source
  visit analyze syn gathEnv : Env -- attribute of first visit
  visit translate inh finEnv : Env -- attributes of second visit
                inh gathInFin : syn.gathEnv  $\sqsubseteq$  inh.finEnv
                syn outcome  : (Errs inh.finEnv)  $\uplus$  (Target inh.finEnv)

itf Target  $\Gamma$  -- interface for Target (parameterized) is not used in the example.

```

As we show later, at the root, we need the value of *gathEnv* to define *finEnv*. This requires *gathEnv* to be placed in a strict earlier visit. Hence we define two visits, ordered by appearance.

Attribute *gathInFin* has a dependent type: it specifies that *gathEnv* is a substring of *finEnv*. A value of this attribute is a proof that essentially states that we did not forget any identifiers.

<sup>4</sup> We may refer to an attribute that is declared earlier (in order of appearance) in the same interface. There is one exception due to the translation to Agda (Section 7.5): in the type of an inherited attribute, we may not refer to synthesized attributes of the same visit.

Similarly, in order to construct *Target* terms, we need to prove that *finEnv* defines the identifiers that occur in the term. In the next section, we construct such proofs by applying data constructors. We may use inherited attributes as *assumptions* and pattern matches against values of attributes as *case distinctions*. Thus, with a dependently typed AG we can formalize and prove correctness properties of our implementation. Agda's type checker validates such proofs using symbolic evaluation driven by unification.

### 7.3.4 Semantics of Attributes

For each production, we give definitions for the declared attributes via rules. At the root, we pass the gathered environment back down as final environment. Thus, these two attributes are equal, and we can trivially prove that the final environment is a substring using either *subRight* or *subLeft*.

```

datasem Root prod root           -- rules for production root of nonterm Root
  top.finEnv      = top.gathEnv      -- pass gathered environment down
  top.gathInFin = subRight {[]}      -- substring proof, using: [] ++ Γ4 ≡ Γ4
  lhs.finEnv     = top.gathEnv      -- pass gathEnv up
  lhs.outcome   = top.outcome      -- pass outcome up

```

For the *use*-production of *Source*, we check if the identifier (terminal *loc.t*) is in the environment. If it is, we produce a *Target* term as value for the outcome attribute, otherwise we produce a *scope* error. For *def*, we introduce an identifier in the gathered environment. No errors can arise, hence we always produce a *Target* term. We prove (*loc.φ*<sub>1</sub>) that the identifier *loc.t* is actually in the gathered environment, and prove (*loc.φ*<sub>2</sub>) using *inSubset* and attribute *lhs.gathInFin* that it must also be in the final environment. For *\_◇\_*, we pass *finEnv* down to both children, concatenate their *gathEnvs*, and combine their *outcomes*.

```

datasem Source           -- rules for productions of Source
prod use
  lhs.gathEnv      = []                -- no names introduced
  lhs.outcome     = with loc.t ∈? lhs.finEnv -- tests presence of t
  | inj1 notIn = inj1 [scope loc.t notIn] -- when not in env
  | inj2 isIn  = inj2 (use loc.t isIn)   -- when in env
prod def
  lhs.gathEnv     = [loc.t]          -- one name introduced
  loc.φ1          = here {loc.t} {syn.lhs.gathEnv} -- proof of t in gathEnv
  loc.φ2          = inSubset lhs.gathInFin loc.φ1 -- proof of t in finEnv
  lhs.outcome     = inj2 (def loc.t loc.φ2) -- never any errors
prod _◇_
  lhs.gathEnv     = left.gathEnv ++ right.gathEnv -- pass names up
  left.finEnv     = lhs.finEnv                -- pass finEnv down
  right.finEnv    = lhs.finEnv                -- pass finEnv down
  left.gathInFin = trans subLeft lhs.gathInFin -- proof for left

```



## 7 AGs with Dependent Types

```

e ::= AGDA [ $\bar{b}$ ]           -- embedded blocks b in AGDA
b ::= i | s | o         -- AGDAX blocks
o ::= inh.c.x | syn.c.x | loc.x -- embedded attribute reference
i ::= itf I  $\bar{x}$ :  $\tau$  v   -- with first visit v, params x, and signature  $\tau$ 
v ::= visit x inh  $\bar{a}$  syn  $\bar{a}$  v -- visit declaration
      |  $\square$                 -- terminator of visit decl. chain
a ::= x: e             -- attribute decl, with Agda type e
s ::= sem: I  $\bar{e}$  t     -- semantics expr, uses interface I  $\bar{e}$ 
t ::= visit x  $\bar{r}$  t   -- visit definition, with next visit t
      |  $\square$                 -- terminator of visit def. chain
r ::= p e'             -- evaluation rule
      | invoke x of c     -- invoke-rule, invokes x on child c
      | child c: I = e -- child-rule, defines a child c, with interface I  $\bar{e}$ 
p ::= o                 -- attribute def
      |  $\{e\}$                 -- Agda dot pattern
      | x  $\bar{p}$                 -- constructor match
e' ::= with e  $\bar{p}$  e'? -- Agda with expression (e' absent when p' absurd)
      | = e                 -- Agda = expression
p' -- Agda LHS
x, I, c -- identifiers, interface names, children respectively
 $\tau$  -- plain Agda type

```

**Figure 7.1:** Syntax of RULER-CORE

A child rule introduces a child with explicit semantics (a value of the type  $T' \text{Source}$ ). Other rules may declare visits and refer to the attributes of the child. An invoke rule declares a visit to a child, and brings the attributes of that visit in scope. Conventional rules define attributes, and may refer to attributes. The dependencies between attributes induces a def-use (partial) order.

Actually, there is one more step to go to end up with a  $\text{AG}_{\text{DA}}^{\text{X}}$  description. A semantics block consists of one or more visit-blocks (in the order specified by the interface), and the rules are partitioned over the blocks. In a block, the *lhs* attributes of that and earlier visits are in scope, as well as those brought in scope by preceding rules. Also, the synthesized attributes of the visit must be defined in the block or in an earlier block. We assign rules to the earliest block that satisfies the def-use order. We convert this partial order into a total order by giving conventional rules precedence over child/invoke rules, and using the order of appearance otherwise:

```

sem  $\diamond$  :  $T' \text{Source} \rightarrow T' \text{Source} \rightarrow T' \text{Source}$  -- signature derived from itf
sem  $\diamond$  locc left locc right = -- semantic function for  $\diamond$ 

```



```

sem : Source                                     -- AGDAX block
visit analyze                                  -- first visit
  child left : Source = loccleft                -- defines a child left
  invoke analyze of left                       -- requires child to be defined
  child right : Source = loccright            -- defines a child right
  invoke analyze of right                     -- requires child to be defined
  syn.lhs.gathEnv = syn.left.gathEnv ++ syn.right.gathEnv

visit translate                                -- second visit
  inh.left.finEnv = inh.lhs.finEnv              -- needs lhs.finEnv
  inh.right.finEnv = inh.lhs.finEnv            -- needs lhs.finEnv
  inh.left.gathInFin = trans ...               -- also needs lhs.gathEnv
  inh.right.gathInFin = trans ...             -- also needs lhs.gathEnv
  invoke translate of left                    -- needs def of inh attrs of left
  invoke translate of right                 -- needs def of inh attrs of right
  syn.lhs.outcome with ...                  -- needs translate attrs of children

```

It is a static error when such an order cannot be satisfied. Another interesting example is the semantic function for the root: it has a child with an interface different from its own, and has two invoke rules in the same visit.

```

sem_root : T' Source → T' Root                  -- semantic function for the root
sem_root locStop =                             -- Source's semantics as parameter
sem : Root visit compile                    -- only one visit
  child top : Source = locctop                -- defines a child top
  invoke analyze of top                      -- invokes first visit of top
  inh.top.finEnv = syn.top.gathEnv            -- passes gathered environment back
  invoke translate of top                   -- invokes second visit of top
  syn.lhs.output = syn.top.gathEnv           -- passes up the gathered env
  syn.lhs.output = syn.top.outcome           -- passes up the result

```

Figure 7.1 shows the syntax of  $AG_{DA}^X$ . In general, interfaces may be parametrized. The interface has a function type  $\tau$  (equal to the type of the nonterminal declaration in  $AG_{DA}$ ) that specifies the type of each parameter, and the kind of the interface (an upper bound of the kinds of the parameters). For an evaluation rule, we either use a with-expression when the value of the attribute is conditionally defined, or use a simple equation as RHS. In the next section, we plug such an expression in a function defined via with-expressions; hence we need knowledge about the with-structure of the RHS.

## 7.5 Translation to Agda

To explain the preprocessing of  $AG_{DA}^X$  to Agda, we give a translation scheme in Figure 7.2 (explained via examples below). This translation scheme is a denotational semantics for  $AG_{DA}^X$ . Also, if the translation is correct Agda, then the original is correct  $AG_{DA}^X$ .

$\llbracket \text{itf } I \bar{x} : \bar{\tau}_x \rightarrow \tau \rrbracket v$	$\rightsquigarrow \llbracket \text{iv } v \rrbracket_{I, \tau}^{\bar{x}; \bar{\tau}_x}$ ; $\llbracket \text{sig } I \rrbracket : \llbracket \tau \rrbracket$ ; $\llbracket \text{sig } I \rrbracket = \llbracket \text{sig } I (\text{name } v) \rrbracket$
$\llbracket \text{iv visit } x \text{ inh } \bar{a} \text{ syn } \bar{b} \text{ v} \rrbracket_{I, \tau}^{\bar{g}}$	$\rightsquigarrow \llbracket \text{iv } v \rrbracket_{I, \tau}^{\bar{g}++\bar{a}++\bar{b}}$ -- interface type for later visits $\llbracket \text{sig } I x \rrbracket : \llbracket \text{at } g_1 \rrbracket \rightarrow \dots \rightarrow \llbracket \text{at } g_n \rrbracket \rightarrow \llbracket \text{resultty } \tau \rrbracket$ $\llbracket \text{sig } I x \rrbracket \llbracket \text{an } g \rrbracket = \llbracket \text{a inh. } a_1 \rrbracket \rightarrow \dots \rightarrow \llbracket \text{a inh. } a_n \rrbracket \rightarrow$ $\llbracket \text{typrod } (\text{syn. } b) (\text{sig } I (\text{name } v)) \rrbracket$
$\llbracket \text{iv } \square \rrbracket_{I, \tau}^{\bar{g}}$	$\rightsquigarrow \llbracket \text{sig } I \square \rrbracket = \square$ -- terminator (some unit-value)
$\llbracket \text{a } x : e \rrbracket$	$\rightsquigarrow \llbracket \text{atname } x \rrbracket : \llbracket e \rrbracket$ -- extract attribute name and type
$\llbracket \text{at } x : e \rrbracket$	$\rightsquigarrow \llbracket e \rrbracket$ -- extract attribute type
$\llbracket \text{an } x : e \rrbracket$	$\rightsquigarrow \llbracket \text{atname } x \rrbracket$ -- extract attribute name
$\llbracket \text{sem } x : I \bar{e} t \rrbracket$	$\rightsquigarrow \llbracket \text{vis lhs } (\text{name } t) \rrbracket \text{ where } \llbracket \text{evt} \rrbracket_I^{\bar{e}, 0}$ -- top of semfun
$\llbracket \text{ev visit } x \bar{r} t \rrbracket_I^{\bar{e}, \bar{g}}$	$\rightsquigarrow \llbracket \text{vis lhs } x \rrbracket : \llbracket \text{sig } I x \rrbracket \llbracket \bar{e} \rrbracket \llbracket \text{an } g \rrbracket$ -- type of visit fun $\llbracket \text{vis lhs } x \rrbracket \llbracket \text{inhs } I x \rrbracket = \llbracket \bar{r} \bar{r} \rrbracket \llbracket \zeta \rrbracket$ -- chain of rules $\llbracket \zeta \rrbracket \rightsquigarrow = \llbracket \text{valprod } (\text{syns } I x) (\text{vis lhs } (\text{name } t)) \rrbracket$ $\text{where } \llbracket \text{evt} \rrbracket_I^{\bar{g}++\bar{a}++\bar{b}}$ -- next visit
$\llbracket \text{ev } \square \rrbracket_I^{\bar{e}, \bar{g}}$	$\rightsquigarrow \llbracket \text{vis lhs } \square \rrbracket : \llbracket \text{sig } I \square \rrbracket \llbracket \bar{e} \rrbracket \llbracket \text{an } g \rrbracket$ ; $\llbracket \text{vis lhs } \square \rrbracket = \square$
$\llbracket \text{r child } c : I = e \rrbracket_k$	$\rightsquigarrow \text{with } \llbracket e \rrbracket \dots \mid \llbracket \text{vis } I (\text{firstvisit } I) \rrbracket \llbracket k \rrbracket$ -- $k$ : remaining rules
$\llbracket \text{r invoke } x \text{ of } c \rrbracket_k$	$\rightsquigarrow \text{with } \llbracket \text{vis } (\text{itf } c) x \rrbracket \llbracket \text{inhs } (\text{itf } c) x \rrbracket$ -- pass inh values $\dots \mid (\text{valprod } (\text{syns } (\text{itf } c) x)) \llbracket k \rrbracket$ -- match syn values
$\llbracket \text{r } p \text{ e}' \rrbracket_k$	$\rightsquigarrow \llbracket \text{ep } e' \rrbracket_p^k$ -- translation for attr def rule
$\llbracket \text{ep with } e \overline{p \text{ e}'} \rrbracket_p^k$	$\rightsquigarrow \text{with } e \dots \mid \llbracket p \rrbracket \llbracket \text{r } p \text{ e}' \rrbracket_k$ -- rule RHS is with-constr
$\llbracket \text{ep } = e \rrbracket_p^k$	$\rightsquigarrow \text{with } e \dots \mid \llbracket p \rrbracket k$ -- rule RHS is expr
$\text{atref inh. } c.x = c_i x$	$\text{atname inh. } x = \text{inh}_a x$ -- naming conventions
$\text{atref syn. } c.x = c_s x$	$\text{atname syn. } x = \text{syn}_a x$ -- <i>atref</i> : ref to attr value
$\text{atref loc. } x = \text{loc}_l x$	$\text{atname } x = x$ -- <i>atname</i> : ref to attr in type
$\text{vis } I x = \text{vis lhs } x$	$\text{sig } I = T' I$ -- <i>vis</i> : name of visit function
$\text{vis } c x = c_v x$	$\text{sig } I x = T' I' x$ -- <i>sig</i> : itf types

 Figure 7.2: Translation of  $\text{AG}_{\text{DA}}^{\text{X}}$  to Agda.

A semantics block in an  $\text{AG}_{\text{DA}}^{\text{X}}$  program is actually an algorithm that makes precise how to compute the attributes as specified by the interface: for each visit, the rules prescribe when to compute an attribute and when to visit a child. The idea is that we map such a block to an Agda function that takes values for its inherited attributes and delivers a dependent product<sup>5</sup> of synthesized attributes. However, such a function would be cyclic: in the presented example, the result  $\text{gathEnv}$  would be needed for as input for  $\text{finEnv}$ . Fortunately, we can bypass this problem: we map to a  $k$ -visit *coroutine* instead. A coroutine is a function that can be invoked  $k$  times. We associate each invocation with a visit of the interface. Values for the inherited attributes are inputs to the invocation. Values for the synthesized attributes are the result of the invocation. In a pure functional language (like Agda), we can encode coroutines as one-shot continuations (or *visit functions* [Saraiva and Swierstra, 1999]).

We generate types for coroutines and for the individual visit functions that make up such a coroutine. These types are derived from the interface. For each visit (e.g. *translate* of *Source*), we generate a type that represents a function type from the attribute types of the inherited attributes for that visit, to a dependent product ( $\Sigma$ ) of the types of the synthesized attributes and the type of the next visit function. These types are parameterized with the attributes of earlier visits (e.g.  $T' \text{Source}' \text{translate } \text{syn}_a \text{gathEnv}$ ). The type of the coroutine itself is the type of the first visit. The type of the last visit is a terminator  $\square$ .

$$\begin{aligned}
T' \text{Source} &= T' \text{Source}' \text{analyze} \\
T' \text{Source}' \text{analyze} &= \Sigma \text{ Env } T' \text{Source}' \text{translate} \\
T' \text{Source}' \text{translate } \text{syn}_a \text{gathEnv} &= \\
&(\text{inh}_a \text{finEnv} : \text{Env}) \rightarrow (\text{inh}_a \text{gathInFin} : \text{syn}_a \text{gathEnv} \sqsubseteq \text{inh}_a \text{finEnv}) \rightarrow \\
&\Sigma (\text{Errs } \text{inh}_a \text{finEnv} \uplus \text{Target } \text{inh}_a \text{finEnv}) \\
&(T' \text{Source}' \square \text{syn}_a \text{gathEnv } \text{inh}_a \text{finEnv } \text{inh}_a \text{gathInFin}) \\
T' \text{Source}' \square \text{syn}_a \text{gathEnv } \text{inh}_a \text{finEnv } \text{inh}_a \text{gathInFin } \text{syn}_a \text{outcome} &= \square
\end{aligned}$$

The restrictions on attribute order in the interface ensure that referenced attributes are in scope. This representation can be optimized a bit by passing only on those attributes that are referenced in the remainder. The scheme for  $\llbracket_{\text{iv}} v \rrbracket_{g, \tau}^I$  formalizes this translation, where  $g$  is the list of preceding attribute declarations, and  $\tau$  the type for  $I$ . The *typrod* function mentioned in the scheme constructs a right-nested dependent product.

The coroutine itself consists of nested continuation functions (one for each visit). Each continuation takes the visit's inherited attributes as parameter, and consists of a tree of with-constructs that represent intermediate computations for computations of attributes and invocations of visits to children. Each leaf ends in a dependent product of the visit's synthesized attributes and the continuation function for the next visit<sup>6</sup>.

$$\begin{array}{ll}
\text{sem} \diamond : T' \text{Source} \rightarrow T' \text{Source} \rightarrow T' \text{Source} & \text{-- example translation for } \diamond \\
\text{sem} \diamond \text{loc}_c \text{left } \text{loc}_c \text{right} = \text{lhs}_v \text{analyze } \underline{\text{where}} & \text{-- delegates to first visit function} \\
\text{lhs}_v \text{analyze} : T' \text{Source}' \text{analyze} & \text{-- signature of first visit function} \\
\text{lhs}_v \text{analyze } \underline{\text{with}} \dots & \text{-- computations for } \text{analyze} \text{ here}
\end{array}$$

<sup>5</sup> A dependent product  $\Sigma \tau f = (\tau, f \tau)$  parameterizes the RHS  $f$  with the LHS  $\tau$ .

<sup>6</sup> As a technical detail, a leaf of the with-tree may also be an *absurd pattern*. These are used in Agda to indicate an alternative that is never satisfiable. A body for such an alternative cannot be given.

## 7 AGs with Dependent Types

```

... = (lhssgathEnv, lhsvtranslate) ahwere           -- result of first visit function
    lhsvtranslate : T' Source' translate lhssgathEnv -- last visit function
    lhsvtranslate lhsifinEnv lhsigathInFin with ... -- computations for translate here
... = (lhssoutcome, lhsv□) where                   -- result of second visit function
    lhsv□ : T' Source' □ lhssgathEnv lhsifinEnv lhsigathInFin lhssoutcome
    lhsv□ = □                                       -- explicit terminator value

```

The scheme  $\llbracket_{\text{ev } \nu} \bar{e}, \bar{g} \rrbracket_I$  formalizes this translation for a visit  $\nu$  of interface  $I$ , where  $\bar{e}$  are type arguments to the interface (empty in the example), and  $\bar{g}$  are the attributes of previous visits.

The with-tree for a visit-function consists of the translation of child-rules, invoke-rules and evaluation rules. Each rule plugs into this tree. For example, the translation for  $\llbracket \text{child left} : \text{Source} = \text{loc}_s \text{left} \rrbracket$  is:

```

... with locsleft           -- evaluate RHS to get first visit fun
... | leftvanalyze with ... -- give it a name + proceed with remainder

```

For  $\llbracket \text{invoke translate of left} \rrbracket$  the translation is:

```

... with leftvtranslate leftifinEnv leftigathInFin -- visit fun takes inh attrs
... | (leftsoutcome, leftvsentinel) with ...      -- returns product of syn attrs

```

For  $\llbracket \text{lhs.gathEnv} = \text{left.gathEnv} ++ \text{right.gathEnv} \rrbracket$ :

```

... with leftsgathEnv ++ rightsgathEnv -- translation for RHS
... | lhssgathEnv with ...           -- LHS + remainder

```

For  $\llbracket \text{lhs.outcome with} \dots \rrbracket$  (where the RHS is a with-construct), we duplicate the remaining with-tree for each alternative of the RHS:

```

... with leftsoutcome           -- translation for RHS
... | inj1 es with rightsoutcome
... | inj1 es1 | inj1 es2 with inj1 (es1 ++ es2) -- alternative one of four
... | inj1 es1 | inj1 es | lhssoutcome with ... -- LHS + remainder
... | inj1 es1 | inj2 - with inj1 es1           -- alternative two of four
... | inj1 es1 | inj2 - | lhssoutcome with ... -- LHS + remainder
... | inj2 ...                                     -- remaining two alternatives

```

The scheme  $\llbracket [r]_k \rrbracket$  formalizes this translation, where  $r$  is a rule and  $k$  the translation of the rules that follow  $r$ .

The size of the translated code may be exponential in the number of rules with with-constructs as RHS. It is not obvious how to treat such rules otherwise. Agda does not allow a with-construct as a subexpression. Neither can we easily factor out the RHS of a rule to a separate function, because the conclusions drawn from the evaluation of preceding rules are not in scope of this function. Fortunately, for rules that would otherwise cause a lot of needless duplication, the programmer can perform this process manually.

When dependent pattern matching brings assumptions in scope that are needed *across* rules, the code duplication is a necessity. To facilitate that pattern matching effects are visible

across rules, we need to ensure that the rule that performs the match is ordered before a rule that needs the assumption. Chapter 3 shows how such non-attribute dependencies can be captured.

The translated code has attractive operational properties. Each attribute is only computed once, and each node is at most traversed  $k$  times.

## 7.6 Partially Defined Attributes

A fine granularity of attributes is important to use an AG effectively. In the mini-compiler of Section 7.3, we could replace the attribute *outcome* with an attribute *code* and a separate attribute *errors*. This would be more convenient, since it would not require a pattern match against the *output* attribute to collect errors. This is convenient in general, as a finer granularity of attributes gives more opportunities to use default rules. However, we cannot produce a target term in the presence of errors, thus *code* would not have a total definition. Therefore, we were forced to combine these two aspects into a single attribute *outcome*. It is common to use partially defined attributes in an AG. This holds especially when the attribute's value (e.g. *errors*) determines if another attribute is defined (e.g. *code*). We present a solution that uses the partitioning of attributes over visits.

The idea is to make the availability of visits dependent on the value of a preceding attribute. We split up the *translate* visit in a visit *report* and a visit *generate*. The visit *report* has *errors* as synthesized attribute, and *generate* has *code*. Furthermore, we enforce that *generate* may only be invoked (by the parent in the AST) when the list of errors reported in the previous visit is empty. We accomplish this with an additional attribute *noErrors* on *generate* that gives evidence that the list of errors is empty. With this evidence, we can give a total definition for *code*.

```

if Source -- Root's visit needs to be split up in a similar way
  visit report  syn errors  :Errs inh.funEnv -- parent can inspect errors
  visit generate inh noErrors :syn.errors ≡ [] -- enforces invariant
                    syn code   :Target inh.funEnv -- only when errors is empty

datasem Source prod use -- example for production use
  loc.testInEnv = loc.t ∈? lhs.funEnv -- scheduled in visit report
  lhs.code with loc.testIn | lhs.noErrors -- scheduled in visit generate
  | inj1 - | () -- cannot happen, hence an absurd pattern
  | inj2 isIn | refl = use loc.t isIn -- extract the evidence needed for the code term

datasem Source prod ◊ -- leftNil: (α : Env) → (β : Env) → (α ++ β ≡ []) → (α ≡ [])
  left.noErrors = leftNil left.errors right.errors lhs.noErrors -- right.noErrors similar
  lhs.code = left.code ◊ right.code -- scheduled in visit generate

```

For this approach to work, it is essential that visits are scheduled as late as possible, and only those that are needed.

Another application of the above idea is related to proofs for special cases, i.e. when we want to prove that with additional assumptions on inherited attributes, the synthesized attributes meet additional criteria. These assumptions are modeled as additional inherited

## 7 AGs with Dependent Types

attributes. However, since we are required to pass values for inherited attributes, our AG would only work for these special cases. For example, suppose that we want to prove that an AG for a type inferencer is complete. To do so, we give a typing derivation as input, and require a proof that the inferred type is more general than the type of the typing derivation. To infer a type, we do not want to provide a typing derivation, and in the proof, we do not want that the typing derivation would be defined in terms of the typing derivation. The typing derivation assumption is thus partially defined: only needed for the proof. Hence, we define the attributes for the proof in a separate visit.

```
itf Expr
  visit infer   inh env : Env syn self : Expr syn errs : Errs
  visit typed   inh  $\phi_1$  : syn.errs  $\equiv []$  syn  $\tau$  : Ty
  visit proof   inh  $\tau'$  : Ty -- special visit for proof
  inh deriv : inh.env  $\vdash$  syn.self : inh. $\tau'$  -- description of typing derivation.
  syn  $\phi_2$  : syn. $\tau \leq$  inh. $\tau'$ 
```

We can generalize the presented approach (Section 7.B) by defining a fixed number of alternative sets of attributes for a visit, and use the value of a preceding attribute to select one of these sets [Middelkoop et al., 2010d].

## 7.7 Related Work

Dependent types originate in Martin-Löf's Type Theory. Several dependently-typed programming languages increasingly gain popularity, including the languages Agda [Norell, 2009], Epigram [McBride, 2004], and Coq [Bertot, 2008]. We present the ideas in this chapter with Agda as host language, because it has a concept of a dependent pattern match, to which we straightforwardly map the left-hand sides of AG rules. Also, in Coq and Epigram, a program is written via interactive theorem proving with tactics or commands. The preprocessor-based approach of this chapter, however, suits a declarative approach more.

Attribute grammars [Knuth, 1968] are considered to be a promising implementation for compiler construction. Recently, many Attribute Grammar systems arose for mainstream languages, such as the systems JastAdd [Ekman and Hedin, 2007] and Silver [Van Wyk et al., 2008] for Java, and UUAG [Löh et al., 1998] for Haskell. These approaches may benefit from the stronger type discipline as presented in this chapter; however, it would require an encoding of dependent types in the host language.

In languages languages with meta-programming facilities, it is sometimes possible to implement AGs without the need of a preprocessor. Viera et al. [2009] show how to implement AGs into Haskell via type level programming. Each rule exposes in its type the attributes that it needs and the attributes that it defines. The rules can be composed via combinators. At one place in the program, the knit point, a proof is constructed that the set of used attributes equals the defined attributes. This proof is subsequently mapped to a semantic function. A combination of that paper with our work, would fit well in Agda, if Agda had a mechanism

similar to Haskell’s class system. Alternatively, it may be possible to embed first-class AGs in Agda, while using a preprocessor to generate boilerplate code.

AGs have a straightforward translation to cyclic functions in a lazy functional programming language [Swierstra and Alcocer, 1998]. To prove that cyclic functions are total and terminating is a non-trivial exercise. Kastens [Kastens, 1980] presented Ordered Attribute Grammars (OAGs). In OAGs, the evaluation order of attribute computations as well as attribute lifetime can be determined statically. Saraiva [Saraiva and Swierstra, 1999] described how to generate (noncyclic) functional coroutines from OAGs. The coroutines we generate are based on these ideas.

## 7.8 Conclusion

We presented  $AG_{DA}$ , a language for ordered AGs with dependently typed attributes: the type of an attribute may refer to the value of another attribute. This feature allows us to conveniently encode invariants in the type of attributes, and pass proofs of these invariants around as attributes. With a dependently typed AG, we write algebras for catamorphisms in a dependently typed language in a composable way. Each attribute describes a separate aspect of the catamorphism.

A particular advantage of composability is that attributes can easily be added and shared. Moreover, via *local attributes* we can specify invariants and proofs at those places where the data is. We prove for the example in Section 7.3 that the final environment must be equal to the gathered environment at the root of the tree:

```
datasem Root prod root -- more rules for the root production
  loc.eqEnvs : inh.top.finEnv  $\equiv$  syn.top.gathEnv -- signature for local attr
  loc.eqEnvs = refl -- proof of the equality
```

The approach we presented is lightweight, which means that we encode AGs as an embedded language (via a preprocessor), such that type checking is deferred to the host language. To facilitate termination checking, we translate the AG to a coroutine (Section 7.5) that encodes a terminating, multi-visit traversal, under the restriction that the AG is ordered and definitions for attributes are total.

The preprocessor approach fits nicely with the interactive Emacs mode of Agda. Type errors in the generated program are traceable back to the source: in a statically checked  $AG_{DA}$  program these can only occur in Agda blocks. These Agda blocks are literally preserved; due to unicode, even attribute references can stay the same. Also, the Emacs mode implements interactive features via markers, which are also preserved by the translation. The AG preprocessor is merely an additional preprocessing step. Not all features integrate seamlessly, however. Syntactical errors in Agda blocks, such as an omitted closing parenthesis, may only be discovered during parsing of the generated code surrounding the block. This can be remedied by validating the syntax of Agda blocks during the preprocessing. A complication arises because the code of rules may occur multiple times in the generated code. Also, the case splitting feature causes the generated program to be transformed, such that it scrutinizes

## 7 AGs with Dependent Types

on a variable chosen by the programmer. The additional equations generated by case splitting need to be transformed back to rules with a with-construct. Fortunately, these are not fundamental problems.

With some generalizations, the work we have presented is a proposal for a more flexible termination checker for Agda that accepts  $k$ -orderable cyclic functions, if the function can be written as a non-cyclic  $k$ -visit coroutine.

As future work, it may be possible to exploit patterns in AG descriptions to generate boilerplate for proofs. For example, we can generate boilerplate code to express termination and monotonicity properties of fixpoint iteration in AGs (Chapter 5), and generate boilerplate code for standard environment lookup and extension patterns. Also, it may be possible to generate proof and AG templates from a type system specification [Dijkstra and Swierstra, 2006b].

### 7.A Implementation of the Support Code

In this section, we give a definition of the support code mentioned briefly in Section 7.3. This section also serves to give a bit more background information about Agda's syntax.

In a dependently typed language, the interpretation of an algebraic data type in an intuitionistic logic is a relation between the type parameters of the data type. A data constructor is an axiom for the relation. A type is a theorem; a value of that type is a proof that the theorem holds.

We defined the following data types and data constructors to work with proofs for environments. We use the  $\_ \in \_$  data type to prove that an identifier is in the environment, and  $\_ \sqsubseteq \_$  to prove that an environment occurs as substring in an environment.

```
data  $\_ \in \_ : Ident \rightarrow Env \rightarrow Set$  where -- member of environment
  here : {  $\iota : Ident$  } {  $\Gamma : Env$  }  $\rightarrow \iota \in (\iota :: \Gamma)$ 
  next : {  $\iota_1 : Ident$  } {  $\iota_2 : Ident$  } {  $\Gamma : Env$  }  $\rightarrow \iota_1 \in \Gamma \rightarrow \iota_1 \in (\iota_2 :: \Gamma)$ 

data  $\_ \sqsubseteq \_ : Env \rightarrow Env \rightarrow Set$  where -- substring of environment
  subLeft : {  $\Gamma_1 : Env$  } {  $\Gamma_2 : Env$  }  $\rightarrow \Gamma_1 \sqsubseteq (\Gamma_1 ++ \Gamma_2)$ 
  subRight : {  $\Gamma_1 : Env$  } {  $\Gamma_2 : Env$  }  $\rightarrow \Gamma_2 \sqsubseteq (\Gamma_1 ++ \Gamma_2)$ 
  trans : {  $\Gamma_1 : Env$  } {  $\Gamma_2 : Env$  } {  $\Gamma_3 : Env$  }  $\rightarrow \Gamma_1 \sqsubseteq \Gamma_2 \rightarrow \Gamma_2 \sqsubseteq \Gamma_3 \rightarrow \Gamma_1 \sqsubseteq \Gamma_3$ 
```

A membership proof for an identifier in the environment states that either the identifier is at the head of the environment, or there is a proof that it is in the tail of the environment. The substring-proof gives prefixes and suffixes to the encapsulated environment that together give the encapsulating environment. Note the use of curly braces here. These represent implicit parameters, which are denoted with similar syntax as implicit parameters for Haskell. An argument for an implicit parameter may be omitted if it can be derived from the context via unifications.

The arrow type constructor can be interpreted as the logical implication. The parameters of a function are assumptions, and the return type is the conclusion. A function thus takes proofs for these assumption as parameter, and transforms these into a proof for the result. The following functions operate on proofs of the above types. When an identifier exists in an



environment, then *append* proves that it also exists in a suffixed version of that environment. Similarly, *prefix* gives a prove for a prefixed environment. Function *inSubset* uses these two to prove that when an identifier occurs in a substring of an environment, it also occurs in the environment itself.

$$\begin{aligned}
& \text{append} : \{ \iota : \text{Ident} \} \{ \Gamma : \text{Env} \} \rightarrow (\iota \in \Gamma) \rightarrow (\Gamma' : \text{Env}) \rightarrow (\iota \in (\Gamma ++ \Gamma')) \\
& \text{append} \{ \iota \} \{ .\iota :: \Gamma \} \text{ (here) } \Gamma' = \text{here} \{ \iota \} \{ \Gamma ++ \Gamma' \} \\
& \text{append} \{ \iota \} \{ nm' :: \Gamma \} \text{ (next in } \Gamma) \Gamma' = \text{next} (\text{append} \{ \iota \} \{ \Gamma \} \text{ in } \Gamma \Gamma') \\
& \text{append} \{ - \} \{ [] \} \quad \quad \quad () \quad \quad \quad - \\
& \text{prefix} : \{ \iota : \text{Ident} \} \{ \Gamma' : \text{Env} \} \rightarrow (\iota \in \Gamma') \rightarrow (\Gamma : \text{Env}) \rightarrow (\iota \in (\Gamma ++ \Gamma')) \\
& \text{prefix in } \Gamma' [] \quad \quad \quad = \text{in } \Gamma' \\
& \text{prefix in } \Gamma' (x :: \Gamma) = \text{next} (\text{prefix in } \Gamma' \Gamma) \\
& \text{inSubset} : \{ \iota : \text{Ident} \} \{ \Gamma : \text{Env} \} \{ \Gamma' : \text{Env} \} \rightarrow (\Gamma' \sqsubseteq \Gamma) \rightarrow \iota \in \Gamma' \rightarrow \iota \in \Gamma \\
& \text{inSubset} (\text{subLeft} \{ - \} \{ \Gamma' \}) \text{ in } \Gamma' = \text{append in } \Gamma' \Gamma' \\
& \text{inSubset} (\text{subRight} \{ \Gamma \}) \quad \quad \quad \text{in } \Gamma' = \text{prefix in } \Gamma' \Gamma \\
& \text{inSubset} (\text{trans subL subR}) \quad \text{in } \Gamma' = \text{inSubset subR} (\text{inSubset subL in } \Gamma')
\end{aligned}$$

In case of *append*, the environment cannot be empty when we have a proof that an identifier occurs in it. However, to satisfy the totality checker, we are required to give a function definition for this case. Since a match against such a pattern cannot succeed, the match is called absurd, and no function body has to be given.

The operator  $\in_?$  takes an identifier  $\iota$  and an environment  $\Gamma$ , and either gives a prove that the identifier is in the environment, or gives a proof that it is not in the environment. The sum type  $\_ \uplus \_$  (named *Either* in Haskell) provides constructors *inj<sub>1</sub>* (*Left* in Haskell) and *inj<sub>2</sub>* (*Right* in Haskell) for this purpose.

For the definition of  $\in_?$ , we use function *notFirst* to prove by contradiction that if an identifier does not occur in the tail of the environment, and is also not equal to the head of the environment, that it is neither in the whole environment.

$$\begin{aligned}
& \text{notFirst} : \{ \iota : \text{Ident} \} \{ nm' : \text{Ident} \} \{ \Gamma : \text{Env} \} \rightarrow \\
& \quad \neg(\iota \equiv nm') \rightarrow \neg(nm' \in \Gamma) \rightarrow \neg(nm' \in (\iota :: \Gamma)) \\
& \text{notFirst } \phi_1 \_ = \lambda \text{ here} \quad \rightarrow \phi_2 \text{ refl} \\
& \text{notFirst } \_ \phi_1 = \lambda (\text{next } \phi_2) \rightarrow \phi_1 \phi_2
\end{aligned}$$

Negation of a type (neg tau) is defined as a function  $\tau \rightarrow \perp$ . The type  $\perp$  (falsum) does not have any data constructors, so we cannot construct it explicitly, but can match against it with an absurd pattern. If we can derive its value, we proved a contradiction between the assumptions we made.

$$\begin{aligned}
& \_ \in_? \_ : (\iota : \text{Ident}) \rightarrow (\Gamma : \text{Env}) \rightarrow \neg(\iota \in \Gamma) \uplus (\iota \in \Gamma) \\
& nm' \in_? [] = \text{inj}_1 \lambda () \\
& nm' \in_? (\iota :: \Gamma) \quad \text{with } \iota \equiv_? nm' \\
& nm' \in_? (.nm' :: \Gamma) \quad | \text{yes refl} = \text{inj}_2 \text{ here} \\
& nm' \in_? (\iota :: \Gamma) \quad \quad | \text{no } \phi' \text{ with } nm' \in_? \Gamma
\end{aligned}$$

## 7 AGs with Dependent Types

$$\begin{array}{l} nm' \in? (\iota :: \Gamma) \quad | \text{no } \phi' \quad | \text{inj}_2 \phi = \text{inj}_2 (\text{next } \{nm'\} \{ \iota \} \phi) \\ nm' \in? (\iota :: \Gamma) \quad | \text{no } \phi' \quad | \text{inj}_1 \phi = \text{inj}_1 (\text{notFirst } \phi' \phi) \end{array}$$

When the environment is empty, the identifier is not in the environment. We thus use  $\text{inj}_1$  and need to construct a negation. This is a function, with an absurd pattern as first parameter, since none of the data constructors of  $\_ \in \_$  can be applied. The other cases apply when the environment is not empty. Also note that patterns in Agda must be strictly linear: there may only be one introduction of an identifier. If there are multiple locations, the identifier must be prefixed with a dot.

In Section 7.6, we used the helper function  $\text{leftNil}$ . It is implemented by case distinction on its first argument. When  $\alpha$  is empty, the requested property trivially holds. When  $\alpha$  is not empty, normalization of  $\alpha ++ \beta$  gives another constructor than  $[]$ , hence the absurd pattern.

$$\begin{array}{l} \text{leftNil} : (\alpha : \text{Env}) \rightarrow (\beta : \text{Env}) \rightarrow (\alpha ++ \beta \equiv []) \rightarrow (\alpha \equiv []) \\ \text{leftNil } [] \quad \_ \text{ refl} = \text{refl} \\ \text{leftNil } (\_ :: \_) \_ \text{ } = () \end{array}$$

### 7.A.1 Absurd Rules

There may be productions for which no semantics for a given interface exists. For example, consider a grammar for a statically sized bit array.

```
grammar BitArray :  $\mathbb{N} \rightarrow \text{Set}$       -- statically sized bit array
prod nil      : BitArray 0           -- empty bit array
prod cons     : BitArray (suc n)    -- non-empty bit array
term? n      :  $\mathbb{N}$                    -- length of the tail
term hd      : Bool                 -- bit at the head of the array
nonterm tl   : BitArray n           -- tail of the bit array
```

We declare a synthesized attribute  $\text{head}$  that stands for the head bit of the array. We can extract this bit when the array is not empty. Hence, we state this requirement as inherited attribute.

```
itf BitArray n visit extract      -- Interface to extract the head
inh prf      : n > 0              -- Proofs that the array is not empty
syn head    : Bool                 -- Must return the head bit
```

If the array is empty, then we cannot give a value for  $\text{lhs.head}$ . Fortunately, the proof helps us out. If the array would be empty, then we would not be able to give the proof. Indeed, we can match with an absurd pattern against the proof, so that we do not have to give a definition for  $\text{lhs.head}$ .

```
datasem BitArray
prod nil     lhs.head with lhs.prf -- would be proof of  $\text{suc } 0 \leq 0$ 
```

```

| () -- not inhabited
prod cons lhs.head = loc.hd -- trivial

```

Under these conditions, a semantics for *nil* cannot be given.

When the interface has multiple attributes, or even multiple visits, then the above code would have to be duplicated for each synthesized attribute of the production, and each inherited attribute of the children. To prevent such code duplication, we introduce an absurd-rule. Its LHS  $p$  must be an absurd pattern that matches against the outcome of the RHS  $e$ .

```

r ::= absurd p = e -- absurd rule (with absurd p)

```

For example, we can use it to match against *lhs.prf* in the *nil* production.

```

datasem BitArray prod nil
absurd () = lhs.prf -- would be proof of  $suc\ 0 \leq 0$ 

```

Attribute definitions and child declarations must be omitted if these would appear later than the absurd-rule in the rule order. The translation is relatively straight-forward.

```

[[r absurd p = e]]k ~~> with e ... | [[p]] -- ends the current with-branch

```

Since each with-branch ends in an absurd pattern, the continuation  $k$  is not needed. Since the follow-up rules on an absurd rule are not generated, we also demand that these are not specified in the first place.

These absurd-rules have consequences for the rule scheduling algorithm. We want the absurd-rules to be scheduled early, such that we can omit the rules that would follow. Also, we want their scheduling to be predictable, such that we know which rules we can and must omit. The scheduling algorithm consists of a number of phases.

- In the first phase, attributes are scheduled to visits of the interface. We assume that this step is performed manually, although it can be automated to a large extent, as is implemented in UUAGC [Löh et al., 1998].
- In the remaining phases, we can deal with each productions independently. For each production we construct a DAG that captures the dependencies between rules (Chapter 3). In this DAG, we identify the (indirect) predecessors of absurd-rules. These predecessors, combined with the absurd-rules, we schedule as early as possible. The partial order imposed by the DAG is turned into a total order by giving precedence first to absurd-rules, then evaluation rules, invoke rules, and finally child rules.
- In the final phase, we schedule the remaining rules as late as possible. Superfluous rules end up in the terminator visit.

If the DAG is non-cyclic, this algorithm properly schedules the rules. The DAG models the def/use dependencies of the rules. Each subsequent pass preserves these dependencies. Thus, the algorithm is sound. Also, if an absurd-rule  $r_1$  could be scheduled earlier than a non-absurd rule  $r_2$ , then  $r_2$  is not a predecessor of  $r_1$ . However, then  $r_1$  would have been scheduled earlier as absurd-rules take precedence.

An absurd-rule thus forces the attributes its RHS refers to, to be scheduled as early as possible. It preferably does not have too complex dependencies (i.e. transitively speaking, only dependencies on inherited attributes and local attributes), so that it is clear when it is scheduled.

## 7.B Dependent Nonterminal Attribution

In Section 7.6, we showed how to deal with an attribute  $p$  that is only defined when another attribute  $q$  has a particular value  $v$ . The trick is to move  $p$  to a later visit than  $q$ , and add the invariant to  $p$ 's visit that requires that it can only be invoked when  $q$  equals  $v$ . This invariant is expressed as additional attribute, which can then also be used in the definition of  $p$ . This is an example of a dependent attribution of nonterminals: depending on the value of  $q$ , the remaining attributes were either all present, or none were present. This approach can be generalized to allow a fixed number of different sets of attributes depending on the value of preceding attributes.

The responsibility for choosing a set of attributes can be given to the caller or the callee. The callee is the node that is a child of the caller. We divide responsibilities as follows. The caller invokes a visit on the callee, and is responsible for selecting one of the alternative interfaces that are offered by the callee. The callee is required to produce results for that choice. The callee can encode restrictions on the available choices for the parent as inherited attributes. The caller must provide values for the inherited attributes of the alternative interface it chooses. With this choice, both the caller and callee can impose demands. The caller imposes these demands by choosing an interface of the callee, and the callee imposes these demands through inherited attributes.

We change the syntax of interfaces to cater for *contexts*. A visit consists of a set of explicitly named contexts  $\bar{z}$ .

```

i ::= itf I  $\bar{x}$  :  $\tau$  v           -- with first visit v, params x, and signature  $\tau$ 
v ::= visit x  $\bar{z}$                  -- visit declaration with a set of contexts (many z)
z ::= context x inh  $\bar{a}$  syn  $\bar{a}$  v -- context x for a visit

```

Notationally, layout becomes important. The contexts  $\bar{z}$  must have the same indentation, and visits and contexts occurring inside a context must have a deeper indentation. As syntactic sugar, we assume that the context-keyword and name may be omitted if there is only one context for a visit. The syntax for the *terminator* visit is not needed anymore. It can be modeled as a visit with zero contexts. The names of contexts must be unique per interface. It allows us to distinguish contexts of different visits from each other, which is needed in  $AG_{DA}$  because of its syntactical conveniences.

For example, we define two contexts for the *generate* visit of the earlier example. The context *errorfree* contains the *code* attribute, but it may only be invoked when errors are absent. The context *haserrors* provides a *pretty* attribute with a pretty print of the program. It does not pose restrictions on the *errors* attribute.

```

itf Source
  visit report syn errors : Errs inh.fnEnv

```

```

visit generate           -- a visit may consist of one or more contexts
context errorfree      -- a context has a name
  inh noErrors : syn.errors  $\equiv$  []
  syn code       : Target inh.finEnv
context haserrors     -- a context may also contain subsequent visits
  syn pretty       : Doc

```

The callee must provide rules for each context. We split a semantics-block  $t$  for a visit up into a context  $u$  for each declared context of that visit in the corresponding interface. Such a context defines the next visit. For visits with more than one context, the caller must explicitly invoke them by means of an invoke rule.

```

t ::= visit  $x \bar{u}$            -- visit definition, with next visit  $t$ 
u ::= context  $x \bar{r} t$       -- context with next visit  $t$ 
r ::= invoke  $x_1$  of  $c$  context  $x_2$  -- modified invoke rule

```

Similarly to the notational conveniences above, we allow the context-keyword and name to be omitted if there is only one context declared for a visit.

For example, for production  $\diamond$ , we choose contexts of the children depending on the context the visit itself is in. In this example, the context for the child is the same as the context of the parent. This is not a requirement.

```

datasem Source prod  $\diamond$ 
  lhs.errors = left.errors ++ right.errors           -- collect errors
  context errorfree                                 -- rules exclusive for errorfree
    invoke generate of left context errorfree        -- explicit invoke
    invoke generate of right context errorfree       -- explicit invoke
    left.noErrors = leftNil left.errors right.errors lhs.noErrors
    left.noErrors = rightNil left.errors right.errors lhs.noErrors
    lhs.code = left.code  $\diamond$  right.code
  context haserrors                                 -- rules exclusive for haserrors
    invoke generate of left context haserrors       -- explicit invoke
    invoke generate of right context haserrors       -- explicit invoke
    lhs.pretty = left.pretty  $\oplus$  right.pretty       -- collect pretty print

```

Ultimately, the choice for the context is made at the root. Either we make this choice external to the AG in terms of the generated coroutine, or use another AG extension [Middelkoop et al., 2010a]: *clauses*. A context may be split further into clauses. A clause may contain special match-rules, which may contain failing pattern matches. When a pattern match fails, execution backtracks to the next clause. The clauses are a means of case distinction on multiple rules at once. This is needed, in order to use different invoke-rules, depending on the values of an attribute.

```

itf Root visit compile syn outcome : (Errs syn.finEnv)  $\uplus$  (Target syn.finEnv)
datasem Root prod root                                     -- some rules omitted

```

## 7 AGs with Dependent Types

```

visit compile                                -- clauses of visit/context
  clause emptyerrors                          -- linear sequence of clauses
    match [] = top.errors                       -- possibly failing test
    top.noErrors = refl                         -- if match succeeds
    invoke generate of top context errorfree    -- take errorfree visit
    lhs.outcome = inj2 top.code
  clause nonemptyerrors                       --  $\neg([\equiv top.errors])$ 
    invoke generate of top context haserrors   -- take haserrors visit
    lhs.outcome = inj1 top.errors

```

Clauses and match rules are confined to the visit/context they are defined in. The clauses must be exhaustive. Within these constraints, match-rules can be scheduled as usual. Match-rules are also scheduled as early as possible, similarly to absurd-rules (Section 7.A.1). There is a good reason to do so: we typically do not have explicit dependencies on a match rule, and its in general better to distinguish cases as soon as possible.

In the  $AG_{DA}^X$  translation, the visit function that corresponds to a visit has a parameter for each attribute of that visit. For the translation of contexts we give an additional, initial parameter to such a visit function. The value is a *handle* [Middelkoop et al., 2010b]: it describes what context we want, and what the type of that context is supposed to be. The handle can be considered a typed version of the control parameter in Kennedy and Warren [1976]. For example, for visit *generate*, there are two contexts, *errorfree* and *haserrors* respectively. For each context, we generate a type that specifies the types of the attributes (as described earlier):

```

T' Source' errorfree ...
T' Source' haserrors ...

```

Each constructor of the handle-type is indexed by one of these context types:

```

data H' Source' generate : (lhssgathEnv : Env) → (lhsserrors : Errs lhssgathEnv) → Set → Set
  where
    errorfree : ∀ {lhssgathEnv} {lhsserrors} →
      H' Source' translate lhssgathEnv lhsserrors (T' Source' errorfree lhssgathEnv lhsserrors)
    haserrors : ∀ {lhssgathEnv} {lhsserrors} →
      H' Source' translate lhssgathEnv lhsserrors (T' Source' haserrors lhssgathEnv lhsserrors)

```

The visit function takes such a handle ( $H'Generate \beta$ ) and returns  $\beta$ . The caller thus knows what  $\beta$  is, and the callee can find this out by pattern matching against the data constructors. The actual type of the visit function, and its implementation becomes:

```

T' Source' generate synagathEnv synaerrors
  = ∀ { $\beta$ } → H' Source' synagathEnv synaerrors generate  $\beta$  →  $\beta$ 
lhsvgenerate : T' Source' generate lhssgathEnv lhsserrors
lhsvgenerate errorfree = ... -- translation for errorfree
lhsvgenerate haserrors = ... -- translation for haserrors

```

The types in question seem rather complex, although this is mainly plumbing to pass attributes around on the type level. Fortunately, AGs alleviate us from writing such code by hand.

The presented mechanism opens up the possibility to have non-linear visit sequences. It also allows us to express interfaces for the ordered AGs as presented by Kennedy and Warren [1976]. Each context can represent an operation or query to be performed on the AST. For example, we could define an interface on the AST of types, such that one context computes the free variables, and another one applies a substitution to the type. The presented approach only permits branching. However, it may be possible to generalize the approach further and allow branches to merge and loop. Merging of branches could be of use for proofs of special cases that span multiple visits (Section 7.6). Also, the interfaces can be seen as *session types* for coroutines.

## 7.C Ideas Transferrable to AG Systems for Haskell

Some of the above ideas carry over to AG systems for Haskell, such as UUAG [Löh et al., 1998]. In a dependently typed AG, attributes can represent both values and types. In Haskell, there is a clear distinction between values and types. In an AG for Haskell, we can make an explicit distinction between attributes that represent types (and have a *kind* as type), and attributes that represent values. The type of a type attribute may not refer to other attributes. The type of a value attribute, however, may refer to a type attribute. Type attributes correspond to quantification. An inherited type attribute corresponds to universal quantification, since the caller can choose its instantiation. A synthesized type attribute corresponds to existential quantification. The callee can choose its type, but the caller cannot make an assumption about it. This mechanism allows us to deal with polymorphism in interfaces.

Currently, UUAG only supports kind star data types. We showed how to deal with parameterized data types, and GADT-style data constructors. These extensions allow AGs to be written for data types with a stronger typing discipline. Also, we can deal with class constraints in constructors by introducing explicit wrappers for dictionaries that we can store as additional fields, e.g.:

```
data DictEq :: * → * where
  DictEq :: Eq a ⇒ DictEq a
```

With such extensions, we can handle even non-regular data types, as long as nonterminals have an outermost type construct described by a grammar declaration.

The handles of Appendix 7.B are conventional GADTs, hence the context-idea can be implemented in Haskell.





## 8 Conclusion

This thesis presented several extensions to attribute grammars. With these extensions, complex type inference algorithms can be expressed. When viewed from a high level perspective, this thesis explored notation that allows units of evaluation in an AG to be made explicit, to be annotated, and to be controlled. More concretely, we regard type inference as the simultaneous construction of a derivation tree and the evaluation of the tree's attributes. In an inference algorithm, attribute evaluation and tree construction take turns. The construction of a derivation tree thus introduces the notion of phasing. Certain attributes must be evaluated to decide on the structure of the tree. We provided several examples, in particular in Chapter 5.

With our extensions, we organize the evaluation of attributes in phases, which makes explicit in what state the tree is before and after the evaluation of a phase. We showed how to exploit this additional information to encode fixpoint computations and search algorithms. Although we focussed on type inference, our extensions are actually making it possible to describe complex tree walking automata.

### 8.1 Addressed Challenges

We briefly restate the challenges as mentioned in the instruction. A declarative specification of a type system is used for formal reasoning and explanation. An inference algorithm describes how to infer admissible types for a program. The former is therefore an implementation of the latter. To show that an inference algorithm is indeed an implementation, it is desirable to have a proof that it is consistent with the type system.

As discussed in Section 1.1, it is a non-trivial exercise to construct an inference algorithm for languages such as Haskell (with extensions), which integrate several cross-cutting type systems. In this setting, a formal specification (if it exists at all) and the implementation are complex, which makes it hard to keep both consistent, let alone proof that this is the case. Moreover, such an implementation changes continuously to support more language features or to use different implementation techniques, thus it is likely to be inconsistent. The apparent inconsistency leaves a big gap between theory and practice. With this thesis, we worked towards a solution where the gap is closed by specifying a formal type systems that contain sufficient algorithmic details to be executable.

A question that arose is what execution model to associate with type rules. Despite the absence of a general inference algorithm that fits any type system (Section 1.2.4), most inference algorithms are a complex composition of standard inference techniques. An executable specification thus describes how to tailor these algorithms to the type system. In this thesis, we zoomed in on the description of such underlying algorithms.

Previous work on the language Ruler posed an initial solution. Ruler is a language in which syntax-directed, algorithmic type rules can be described, which in combination with

## 8 Conclusion

some additional annotations can be translated automatically to AGs (Section 1.4). Type rules coincide with productions, and metavariables with attributes. The inference algorithm is thus expressed as the evaluation of AGs.

For a declaratively specified type system, type inference for declarative aspects boils down to inferring the structure of the derivation tree and inferring instantiations for non-functionally constrained meta variables. Higher-order AGs withstanding, AGs assume an apriori fixed tree, and require functional definitions for attributes. Therefore, an inference algorithm for declarative type rules cannot be obtained via a straightforward translation to attribute grammars. We addressed this issue in this thesis by extending the AG language to cater for several inference strategies, and mechanism to combine these strategies.

## 8.2 Solutions

As initial setting, we showed that we can model an AG on type derivation trees instead of parse trees. To deal with non-functional attributes such as types, we used a conventional unification-based strategy with an additional threaded substitution attribute (Section 1.3.11), and expressed unifications in the AG in a declarative manner using higher-order children (Section 1.3.12). The substitution in combination with placeholders in types makes the relations on types functional. Moreover, the implementation of the higher-order child expresses the standard unification algorithm, and the threading of the substitution its algorithmic coordination, since the threading defines the relative order of the unifications and other operations that depend on the substitution. To this setting, we made various improvements which we describe in the remainder of this section.

The explicit threading of the substitution is a tedious job. Alternatively, we can regard unification as an operation with side effects that affects the substitution. In Chapter 3, we showed how to make the visit order explicit and use this order to declaratively specify the relative order of the operations with side effects in addition to the usual order constraints induced by attribute dependencies. To retain the desired referential transparency of attributes in the programming model, side effects may only be used in the construction of higher-order children. In particular, this approach allows AGs to be integrated in a compiler that uses inference monads.

The relative order of operations that provide fresh placeholders and perform unification is largely irrelevant. In Chapter 4, we presented commutable rules for threaded attributes to both model side effects and to relax the order induced by attribute dependencies. In Chapter 4, we abstracted from visits to phases, where a phase corresponds to one or more inferred visits, and allowed us to decouple the actual AG evaluation algorithm. Visits provide a fine-grained model to describe aspects of the evaluation of AGs, whereas with phases we can specify properties of larger chunks of evaluation. We also showed how to encode the Kennedy-Warren AG evaluation algorithm in a strongly-typed functional language.

We exploited the notion of a visit to express typical inference algorithms. In Chapter 5, we presented how to express fixpoint iteration by iterating visits. An invocation of a visit on a subtree specifies how to compute the visit's inherited attributes from its synthesized attributes of the previous iteration. In contrast to conventional fixpoint evaluators for AGs, the notion

of visits allows us to compute stop conditions as synthesized attribute, and have visit-local chained attributes that retain computed values of previous visits.

When an attribute contains an unconstrained placeholder, residuation is a strategy that defers a dependent computation until the placeholder is sufficiently constrained. In Chapter 5, we presented an example that implements the residuation strategy by decoupling a child at one location in the tree and provide input at another location in the tree. This mechanism also provides an integration with constraints (Section 5.2.7).

In Chapter 5, we also presented how to infer the derivation tree as a function of the inherited attributes. In a case study [Middelkoop, 2011a], we illustrated the need for search strategies to infer the structure of derivation trees when the structure of the derivation tree is not functionally defined. Chapter 6 showed how to encode such search strategies.

Finally, our work provides solutions in other contexts than type inference. We illustrated this in the extended edition of this thesis [Middelkoop, 2011b], where we applied our work to graphs. Since many analyses in compilers are based on control-flow or data-flow graphs, we showed how to associate a semantics with each node of such a graph with an AG, while explicitly specify visits to these nodes with a traversal algorithm. Moreover, this approach allows the encoding of reference attributes in an ordered AG.

In Chapter 7, we applied our work to dependently-typed languages. With dependently-typed attributes, invariants between attributes can be expressed, and proven to hold, which allows formal reasoning with attribute grammars. Moreover, the type attributes provide a mechanism for the universal and existential quantification of the type of a semantic functions.

## 8.3 Remarks

We put great effort in ensuring that our extensions retain the ease of composition as offered by AGs so that attributes and rules can still be defined in an aspect-wise and order-independent fashion. Also, our ideas are conservative extensions of AGs. A conventional AG can be expressed straightforwardly, which allows features of RulerCore to be retrofitted on an implementation using conventional AGs.

However, to exploit the visit order, we need to specify in which order the children are visited. For conventional AGs, the traversal over the AST and order of evaluation of rules does not need to be specified, which has the advantage that it is no concern for the programmer. This is only a small price to pay. Such orders are declaratively specified (Chapter 3), and only where needed.

## 8.4 Implementations

We implemented several prototypes to experiment with the extensions presented in this thesis. We validated that our extensions have reasonably efficient implementations. In particular, the ideas are implementable in Haskell. Our prototypes depend on language features such as lazy evaluation, monads and GADTs. However, the underlying ideas themselves are implementable in mainstream languages and other AG systems.

## 8 Conclusion

To our attribute grammar system UUAG, we added higher-order children (Section 1.3.7) and stepwise evaluation (Chapter 6). In essence, we added only features that do not conflict with the conventional notion of AGs. Also, we implemented several features to support the use of AGs in large compiler implementations. In particular, we organized the code generated by UUAG such that separate compilation, debugging, and profiling is possible.

In the tool `ruler-core`, we implemented the programming model with explicit visits (Chapter 3). We experimented with clauses, fixpoint iteration, and other features that benefit from the notion of a visit (Chapter 5). As further case study, `ruler-core` was used in a master project to implement the inference algorithm of the HML type system [Leijen, 2009].

In the tool `ruler-interpreter` we implemented the operational semantics as outlined in Chapter 4. The interpreter provides custom judgment syntax, aspect-weaving of type rules, and a built-in efficient unification mechanism. The simplicity of an interpreter facilitated rapid prototyping with some of RulerCore's features.

## 8.5 Future Work

We implemented and validated our ideas with several prototype implementations. We provided powerful building blocks for the description of inference algorithms with AGs, and the description of patterns that often occur in these contexts. However, to fully exploit our techniques, an integrated implementation of all extensions is needed.

The features that we implemented in UUAG were used to improve the UHC implementation. Moreover, we used UHC as motivation to investigate AG extensions using RulerCore for prototyping purposes. We claim that we provide sufficient expressive power to implement unification and context reduction more concisely in UHC, but to actually do so remains as future work.

We provide the algorithmic underpinning of inference algorithms for AGs. This takes us one step closer to our ultimate goal to derive type inference algorithms from type system specifications. However, the remaining challenges as mentioned in Section 1.5 need to be addressed as well, such as special syntax and first-class abstractions for AGs. Moreover, visits are a prominent component in our current work. A direction of future work is to *infer* properties of visits from a more abstract specification.

A question that remains open is how we can formally prove and ensure properties of the implementations that we generate. We address this topic briefly in Chapter 7. A direction of future work is to generate boilerplate code to support typical proofs.

This thesis provides a core language for inference algorithm descriptions, which paves the way for high-level abstractions of inference algorithms, thus facilitates more complex language implementations, and ultimately leads to software of higher quality.

## Bibliography

- A. V. Aho and J. D. Ullman. Translations on a Context Free Grammar. In *STOC '69*, pages 93–112, 1969.
- J. Aldrich, R. J. Simmons, and K. Shin. SASyLF: an Educational Proof Assistant for Language Theory. In *FDPE '08*, pages 31–40, 2008.
- A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
- B. Arbab. Compiling Circular Attribute Grammars into Prolog. *IBM Journal on Research and Development*, 30:294–309, 1986.
- I. Attali and D. Parigot. Integrating Natural Semantics and Attribute Grammars : the Minotaur System. Technical Report RR-2339, INRIA, 1994.
- Y. Bertot. Coq in a Hurry. *CoRR '06*, 2006.
- Y. Bertot. A Short Presentation of Coq. In *TPHOLs '08*, pages 12–16, 2008.
- R. S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, 21:239–250, 1984.
- A. Bove and P. Dybjer. Dependent Types at Work. In *Language Engineering and Rigorous Software Development*, volume 5520, pages 57–99, 2009.
- J. T. Boyland. Conditional Attribute Grammars. *ACM Transactions on Programming Languages and Systems*, 18(1):73–108, 1996.
- E. C. Brady. IDRIS: Systems Programming meets Full Dependent Types. In *PLPV '11*, pages 43–54, 2011.
- B. Braßel, S. Fischer, M. Hanus, and F. Reck. Transforming Functional Logic Programs into Monadic Functional Programs. In *WFLP'10*, 2010.
- M. Bravenboer, R. Vermaas, J. J. Vinju, and E. Visser. Generalized Type-Based Disambiguation of Meta Programs with Concrete Object Syntax. In *GPCE '05*, pages 157–172, 2005.
- C. Chambers and G. T. Leavens. Typechecking and Modules for Multi-Methods. In *OOPSLA '94*, pages 1–15, 1994.
- J. Cheney and R. Hinze. First-Class Phantom Types. Technical report, Cornell University, 2003.

## Bibliography

- N. Chomsky. Three Models for the Description of Language. *Transactions on Information Theory*, 2:113–124, 1956.
- E. de Vries, R. Plasmeijer, and D. M. Abrahamson. Uniqueness Typing Simplified. In *IFL '07*, pages 201–218, 2007.
- T. Despeyroux. TYPOL : a Formalism to Implement Natural Semantics. Technical Report RT-0094, INRIA, 1988.
- V. Diekert and Y. Métivier. Partial Commutation and Traces. In *Handbook of Formal Languages*, pages 457–533. Springer-Verlag, 1997.
- A. Dijkstra. *Stepping through Haskell*. PhD thesis, Universiteit Utrecht, 2005.
- A. Dijkstra and D. S. Swierstra. Exploiting Type Annotations. Technical report, Universiteit Utrecht, 2006a.
- A. Dijkstra and S. D. Swierstra. Typing Haskell with an Attribute Grammar. In *AFP '04*, pages 1–72, 2004.
- A. Dijkstra and S. D. Swierstra. Ruler: Programming Type Rules. In *FLOPS '06*, pages 30–46, 2006b.
- A. Dijkstra, G. van den Geest, B. Heeren, and S. D. Swierstra. Modelling Scoped Instances with Constraint Handling Rules, 2007.
- A. Dijkstra, A. Middelkoop, and S. D. Swierstra. Efficient Functional Unification and Substitution, 2008.
- A. Dijkstra, J. Fokker, and S. D. Swierstra. The Architecture of the Utrecht Haskell Compiler. In *Haskell Symposium*, pages 93–104, 2009.
- T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. In *OOPSLA '07*, pages 1–18, 2007.
- J. Engelfriet and G. Filé. Passes, Sweeps, and Visits in Attribute Grammars. *Journal of the ACM*, 36(4):841–869, 1989.
- J. Engelfriet and H. J. Hoogeboom. Tree-Walking Pebble Automata. In *Jewels are Forever*, pages 72–83, 1999.
- L. Erkök and J. Launchbury. Recursive Monadic Bindings. In *ICFP '00*, pages 174–185, 2000.
- R. Farrow. Sub-Protocol-Evaluators for Attribute Grammars. *Sigplan Notices*, 19:70–80, 1984.
- R. Farrow. Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars. In *CC '86*, pages 85–98, 1986.

- R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation. In *POPL '92*, pages 223–234, 1992.
- K.-F. Faxén. A Static Semantics for Haskell. *JFP*, 12(4&5):295–357, 2002.
- S. Fischer, O. Kiselyov, and C. Shan. Purely Functional Lazy Non-deterministic Programming. In *ICFP '09*, pages 11–22, 2009.
- J. Fokker and S. D. Swierstra. Abstract Interpretation of Functional Programs using an Attribute Grammar System. *ENTCS*, 238(5):117–133, 2009.
- T. Frühwirth. Theory and Practice of Constraint Handling Rules. *JLP*, 37(1-3):95–138, 1998.
- E. M. Gagnon and L. J. Hendren. SableCC, an Object-Oriented Compiler Framework. In *TOOLS (26)*, pages 140–154, 1998.
- E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *ECOOP '93*, pages 406–431, 1993.
- S. Glesner and W. Zimmermann. Using Many-Sorted Natural Semantics to Specify and Generate Semantic Analysis, 1998.
- J. Hage, S. Holdermans, and A. Middelkoop. A Generic Usage Analysis with Subeffect Qualifiers. In *ICFP '07*, pages 235–246, 2007.
- M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *JLP*, 19/20:583–628, 1994.
- R. Harper. *Practical Foundations for Programming Languages*, 2010.
- B. Heeren, J. Hage, and S. D. Swierstra. Scripting the Type Inference Process. In *ICFP '03*, pages 3–13, 2003a.
- B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for Learning Haskell. In *Haskell Workshop*, pages 62 – 71, 2003b.
- B. J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, 2005.
- P. Heidegger, A. Bieniusa, and P. Thiemann. DOM Transactions for Testing JavaScript. In *TAICPART '10*, pages 211–214, 2010.
- R. Hinze. Deriving Backtracking Monad Transformers. In *ICFP '00*, pages 186–197, 2000.
- P. Hudak, S. L. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. H. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, R. B. Kieburtz, R. S. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language. *SIGPLAN Notices*, 27(5):1–164, 1992.
- G. Huet. The Zipper. *JFP*, 7:549–554, 1997.

## Bibliography

- J. Hughes. Programming with Arrows. In *AFP '04*, pages 73–129, 2004.
- R. J. M. Hughes and S. D. Swierstra. Polish Parsers, Step by Step. In *ICFP '03*, pages 239–248, 2003.
- T. Jim. What Are Principal Typings and What Are They Good For? In *POPL '96*, pages 42–53, 1996.
- L. G. Jones. Efficient Evaluation of Circular Attribute Grammars. *ACM Transactions on Programming Languages and Systems*, 12(3):429–462, 1990.
- M. P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In *AFP '95*, pages 97–136, 1995.
- M. Jourdan and D. Parigot. Internals and Externals of the FNC-2 Attribute Grammar System. In *AGAS '91*, pages 485–504, 1991.
- T. Kaminski and E. Van Wijk. Integrating Attribute Grammar and Functional Programming Language Features (to appear), 2011.
- U. Kastens. Ordered Attributed Grammars. *Acta Informatica*, 13:229–256, 1980.
- L. C. L. Kats, A. M. Sloane, and E. Visser. Decorated Attribute Grammars: Attribute Evaluation Meets Strategic Programming. In *CC '09*, pages 142–157, 2009.
- K. Kennedy and S. K. Warren. Automatic Generation of Efficient Evaluators for Attribute Grammars. In *POPL '76*, pages 32–49, 1976.
- O. Kiselyov. Iteratee IO: Safe, Practical, Declarative Input Processing, 2008.
- O. Kiselyov, C. Shan, D. P. Friedman, and A. Sabry. Backtracking, Interleaving, and Terminating Monad Transformers (functional pearl). In *ICFP '05*, pages 192–203, 2005.
- A. Klaiber and M. Gokhale. Parallel Evaluation of Attribute Grammars. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):206–220, 1992.
- D. E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2): 127–145, 1968.
- D. E. Knuth. Semantics of Context-free Languages: Correction. *Theory of Computing Systems*, 5:95–96, 1971.
- D. E. Knuth. The Genesis of Attribute Grammars. In *WAGA '90*, pages 1–12, 1990.
- M. F. Kuiper and J. Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In *CC '98*, pages 298–301, 1998.
- M. F. Kuiper and S. D. Swierstra. Parallel Attribute Evaluation: Structure of Evaluators and Detection of Parallelism. In *WAGA '90*, pages 61–75, 1990.



- D. Leijen. Flexible Types: Robust Type Inference for First-Class Polymorphism. In *POPL '09*, pages 66–77, 2009.
- M. Y. Levin and B. C. Pierce. TinkerType: a Language for Playing with Formal Systems. *JFP*, 13(2):295–316, 2003.
- A. Löh, A. I. Baars, and D. S. Swierstra. Homepage of the Universiteit Utrecht Attribute Grammar System. <http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem>, 1998.
- E. Magnusson and G. Hedin. Circular Reference Attributed Grammars - their Evaluation and Applications. *SCP '07*, 68(1):21–37, 2007.
- E. Magnusson, T. Ekman, and G. Hedin. Extending Attribute Grammars with Collection Attributes—Evaluation and Applications. *SCAM '07*, 0:69–80, 2007.
- C. D. Marlin. *Coroutines: A Programming Methodology, a Language Design and an Implementation*, volume 95. Springer-Verlag, 1980.
- C. McBride. Epigram: Practical Programming with Dependent Types. In *AFP '04*, pages 130–170, 2004.
- E. Meijer and J. Jeuring. Merging Monads and Folds for Functional Programming. In *AFP*, volume 925, pages 228–266, 1995.
- A. Middelkoop. Case Study with GADTs. In *Inference of Program Properties with Attribute Grammars, Revisited (extended edition)*. Universiteit Utrecht, 2011a.
- A. Middelkoop. AGs on Graphs. In *Inference of Program Properties with Attribute Grammars, Revisited (extended edition)*. Universiteit Utrecht, 2011b.
- A. Middelkoop. AGs with Side Effects (Appendices). In *Inference of Program Properties with Attribute Grammars, Revisited (extended edition)*. Universiteit Utrecht, 2011c.
- A. Middelkoop, A. Dijkstra, and S. D. Swierstra. A Leaner Specification for Generalized Algebraic Data Types. In *TFP*, volume 9, pages 65–80, 2008.
- A. Middelkoop, A. Dijkstra, and S. D. Swierstra. Iterative Type Inference with Attribute Grammars. In *GPCE '10*, pages 43–52, 2010a.
- A. Middelkoop, A. Dijkstra, and S. D. Swierstra. Visit Functions for the Semantics of Programming Languages, 2010b.
- A. Middelkoop, A. Dijkstra, and S. D. Swierstra. Attribute Grammars with Side Effect. In *HOSC*, 2010c.
- A. Middelkoop, A. Dijkstra, and S. D. Swierstra. Visit Functions for the Semantics of Programming Languages. In *WGT '10*, 2010d.

## Bibliography

- A. Middelkoop, A. Dijkstra, and S. D. Swierstra. Stepwise Evaluation of Attribute Grammars (extended version), 2010e.
- A. Middelkoop, A. Dijkstra, and S. Doaitse Swierstra. Visitor-based Attribute Grammars with Side Effect. *ENTCS*, 264:47–69, 2011a.
- A. Middelkoop, A. Dijkstra, and S. Swierstra. A Lean Specification for GADTs: System F with First-Class Equality Proofs. *HOSC*, pages 1–22, 2011b.
- M. d. Mol, M. C. J. D. v. Eekelen, and M. J. Plasmeijer. Theorem proving for functional programmers. In *IFL '02*, pages 55–71, 2002.
- F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
- U. Norell. Dependently-Typed Programming in Agda. In *TLDI '09*, pages 1–2, 2009.
- C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
- B. C. D. S. Oliveira, M. Wang, and J. Gibbons. The Visitor Pattern as a Reusable, Generic, Type-safe Component. In *OOPSLA '08*, pages 439–456, 2008.
- B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly Media, Inc., 2008.
- J. Paakki. PROFIT: A System Integrating Logic Programming and Attribute Grammars. In *PLILP '91*, pages 243–254, 1991.
- J. Palsberg and C. B. Jay. The Essence of the Visitor Pattern. In *COMPSAC '98*, pages 9–15, 1998.
- R. Paterson. A New Notation for Arrows. In *ICFP '01*, pages 229–240, 2001.
- B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.
- K.-J. Rähkä and M. Saarinen. Testing Attribute Grammars for Circularity. *Acta Informatica*, 17:185–192, 1982.
- J. C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *HOSC*, 11(4):363–397, 1998.
- S. J. Russell, P. Norvig, J. F. Candy, J. M. Malik, and D. D. Edwards. *Artificial Intelligence: a Modern Approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- J. Saraiva. Component-Based Programming for Higher-Order Attribute Grammars. In *GPCE '02*, pages 268–282, 2002.
- J. Saraiva and S. D. Swierstra. Purely Functional Implementation of Attribute Grammars. Technical report, Universiteit Utrecht, 1999.
- M. M. Schrage and J. T. Jeuring. Proxima - A Presentation-Oriented Editor for Structured Documents, 2004.

- C. Schürmann. The Twelf Proof Assistant. In *TPHOLs '09*, pages 79–83, 2009.
- P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective Tool Support for the Working Semanticist. In *ICFP '07*, pages 1–12, 2007.
- A. M. Sloane, L. C. L. Kats, and E. Visser. A Pure Object-Oriented Embedding of Attribute Grammars. *ENTCS*, 253:205–219, 2010.
- S. D. Swierstra. Combinator Parsing: A Short Tutorial. In *Language Engineering and Rigorous Software Development*, volume 5520, pages 252–300, 2009.
- S. D. Swierstra and P. R. A. Alcocer. Attribute Grammars in the Functional Style. In *Systems Implementation 2000*, pages 180–193, 1998.
- U. Utrecht. Mini Projects Compiler Construction. <http://www.cs.uu.nl/wiki/bin/view/Cco/MiniProjects>, 2010.
- M. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In *CC '02*, pages 143–158, 2002.
- M. C. J. D. van Eekelen, S. Smetsers, and M. J. Plasmeijer. Graph Rewriting Semantics for Functional Programming Languages. In *CSL '96*, pages 106–128, 1996.
- E. Van Wyk and L. Krishnan. Using Verified Data-Flow Analysis-based Optimizations in Attribute Grammars. *ENTCS*, 176:109–122, 2007.
- E. Van Wyk, O. D. Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. In *CC '02*, pages 128–142, 2002.
- E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an Extensible Attribute Grammar System. *ENTCS*, 203(2):103–116, 2008.
- M. Viera, S. D. Swierstra, and W. Swierstra. Attribute Grammars Fly First-Class: how to do Aspect Oriented Programming in Haskell. In *ICFP '09*, pages 245–256, 2009.
- H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher-Order Attribute Grammars. In *PLDI '89*, pages 131–145, 1989.
- H. Vogt, S. D. Swierstra, and M. F. Kuiper. Efficient Incremental Evaluation of Higher order Attribute Grammars. In *PLILP*, pages 231–242, 1991.
- D. Vytiniotis, S. Weirich, and S. L. Peyton Jones. FPH: First-Class Polymorphism for Haskell. In *ICFP '08*, pages 295–306, 2008.
- M. J. Walsteijn and M. F. Kuiper. Attribute Grammars in Prolog, 1986.
- J. A. Walz. *Extending Attribute Grammar and Type Inference Algorithms*. PhD thesis, Cornell University, 1989.

## *Bibliography*

- S. Wang and D. Ye. On Parallel Evaluation of Ordered Attribute Grammars. *Journal of Computer Science and Technology*, 6:347–354, 1991.
- S. K. Warren. *The Coroutine Model of Attribute Grammar Evaluation*. PhD thesis, Rice University, Houston, TX, 1976.
- J. B. Wells. Typability and Type Checking in System F are Equivalent and Undecidable. *Annals of Pure and Applied Logic*, 98(1-3):111 – 156, 1999.
- M. Wenzel, L. C. Paulson, and T. Nipkow. The Isabelle Framework. In *TPHOLS '08*, pages 33–38, 2008.
- D. Yeh and U. Kastens. Improvements of an Incremental Evaluation Algorithm for Ordered Attribute Grammars. *SIGPLAN Notices*, 23(12):45–50, 1988.

## Samenvatting

Computers zijn niet meer uit onze maatschappij weg te denken. Computerprogramma's worden steeds omvangrijker. De keerzijde is dat deze programma's meer tijd kosten om te maken en te testen, en bovendien ook meer fouten bevatten. Helaas ervaren we regelmatig de consequenties ervan. Autofabrikanten roepen bijvoorbeeld auto's terug om een software-update door te voeren, omdat door fouten in de programmering de auto's op hol konden slaan. Ook fouten in web-browsers worden door duistere figuren uitgebuit om zogenaamde *keyloggers* op computers te installeren die toetsaanslagen afluisteren om uiteindelijk bankrekeningen te plunderen. Oplossingen voor dergelijke problemen kunnen deels in programmeertalen gezocht worden. Onderzoek naar programmeertalen helpt om zowel de onderhoudskosten en de kwaliteit van software te verbeteren.

Met geavanceerde programmeertalen is het mogelijk om computerprogramma's van hoge kwaliteit te maken. Daarvoor is een belangrijk stuk gereedschap van belang: de *compiler*. Een compiler zet een programma wat een programmeur geschreven heeft om in machine-instructies die door de computer uitgevoerd kunnen worden. Hoe geavanceerder de programmeertaal, hoe lastiger het is om een compiler te maken.

Met attributengrammatica's kunnen compilers op een aantrekkelijke manier gemaakt worden. Programmeertalen met ingewikkelde typesystemen zijn echter lastig met attributengrammatica's te schrijven. In dit proefschrift beschouwen we uitbreidingen om attributengrammatica's ook voor het ontwikkelen van compilers voor ingewikkelde programmeertalen in te kunnen zetten.

**Programmeertalen en compilers.** Een computerprogramma verwerkt gegevens die zich in het geheugen van de computer bevinden. In een programmeertaal wordt deze verwerking beschreven. Voor dergelijke beschrijvingen stelt een programmeertaal elementaire verwerkingstaken ter beschikking. Tijdens de uitvoering van het programma krijgt zo'n taak gegevens uit het geheugen van de computer als invoer en laat het resultaten in het geheugen van de computer achter. Bekeken vanaf een hoog niveau kunnen we zeggen dat programmeren het samenstellen van verwerkingstaken is door de uitvoer en invoer van verwerkingstaken aan elkaar te knopen.

Een programmeertaal biedt abstractiemechanismen aan om dergelijke samenstellingen te beschrijven. Programmeren is het opstellen van zo'n beschrijving: de *broncode*. Een *compiler* vertaalt broncode naar instructies die door een computer uitgevoerd kunnen worden: de *machinecode*. Een compiler handelt details af zoals hoe de invoer en uitvoer van taken in het geheugen van de computer gerepresenteerd zijn. In de broncode is het niet nodig om dergelijke details te specificeren, wat het gemak om verwerkingstaken samen te stellen ten goede komt.

De mate waarin een programmeur een verwerkingstaak uit deeltaken kan samenstellen is

## Samenvatting

van grote invloed op de kwaliteit van een programma, en de tijd die het kost om het programma te ontwikkelen en te onderhouden. Wanneer de broncode overzichtelijk is, worden er minder fouten gemaakt. Bovendien hoeft de broncode van een deeltaak maar een keer geschreven te worden. De mate waarin een programmeertaal abstractie van details toestaat speelt hierbij een belangrijke rol.

Programmeertalen bieden vaak voor specifieke domeinen speciale abstractiemechanismen aan. De taal SQL voor het raadplegen van databases is hier een goed voorbeeld van. In SQL beschrijft men het combineren van informatie uit tabellen, terwijl van de representatie van de gegevens in de tabellen en van de volgorde van het combineren geabstraheerd wordt. Idealiter richt de programmeur zich op het totaalplaatje, terwijl de compiler voor een correcte invulling van de details zorgt, eventueel aan de hand van wat expliciete hints die door de programmeur gegeven worden. Met behulp van een relatie tussen broncode en machine-instructies kan dit gedrag gespecificeerd worden.

Een programmeertaal stelt bovendien eisen aan de broncode. Bijvoorbeeld, in een samenstelling van taken dient iedere taak een correct gestructureerde invoer te hebben. Een *type* is een beschrijving van de structuur van een waarde. Ofwel, de invoer dient het juiste type te hebben. De compiler controleert als onderdeel van het vertaalproces of de broncode inderdaad aan deze eisen voldoet, en vormt dus een implementatie van het typesysteem. Een *statische semantiek* in de vorm van een *type system* specificeert deze eisen met een relatie tussen broncode en typen. Het afdwingen van deze eisen voorkomt dat bepaalde (triviale) fouten tijdens de uitvoering van het programma op kunnen treden.

Als onderdeel van het vertaalproces controleert de compiler of de broncode aan de eisen voldoet door een *bewijs* af te leiden dat de broncode relateert aan een type. De machine-instructies worden verkregen door een bewijs af te leiden dat een relatie legt met machine-instructies. Als dit niet lukt, dan bevat het programma een *statische fout* en is het programma ongeldig. Relaties in een semantiek worden doorgaans met afleidingsregels gedefinieerd. Het afleiden van zo'n bewijs wordt *inferentie* of *afleiden* genoemd. De bewijzen hebben een boomstructuur waarin de toepassing van afleidingsregels zichtbaar is.

Vrijheid in het bepalen van het bewijs geeft de compiler de mogelijkheid om details in te vullen. Echter, er bestaan harde theoretische grenzen aan wat voor bewijzen er automatisch afgeleid kunnen worden. Door de taal ingewikkelder te maken, kan er op een hoger niveau geredeneerd worden. Dan is het mogelijk een programma duidelijker uit te drukken, zodat de broncode meer structuur heeft, en er meer aannames zijn om het bewijs mee rond te krijgen. Een direct gevolg is dat de compiler daardoor lastiger wordt om te maken.

**Attributengrammatica's.** Als initiële stap ontleedt een compiler de broncode aan de hand van de grammatica van de programmeertaal. Het resultaat is een boomstructuur, de *abstracte syntaxboom* (AST), dat een expliciete representatie is van de compositionele structuur van de broncode. Deze boomstructuur is geschikt voor syntax-gestuurde vertaling. In dit geval heeft een semantiek een afleidingsregel voor ieder stukje syntax. De structuur van een bewijs komt dan vrijwel overeen met de AST.

Een compiler is ook een computerprogramma, en worden in een programmeertaal geschreven. Attributengrammatica's (AG's) zijn een domein-specifieke programmeertaal voor het uitdrukken van eigenschappen van ASTs, en daarmee dus ook het afleiden van bewijzen voor

relaties van een syntax-gestuurde semantiek. Een AG relateert attributen met elke knoop in de AST, en specificeert functies die waarden van attributen berekenen aan de hand van waarden van andere attributen van een knoop en kinderen van de knoop. De attributen stellen eigenschappen van de broncode voor, en zijn aspecten of *getuigen* van het bewijs, zoals typen, lijsten van instructies en foutmeldingen.

De voordelen van AG's ten opzichte van algemene programmeertalen zijn dat niet beschreven wordt *hoe* de AST afgelopen wordt. Daardoor kunnen de berekeningen van attributen in afzondering beschreven worden, wat vele voordelen biedt in termen van hergebruik, overzicht en documentatie. De samenstelling van deze berekeningen wordt automatisch bepaald. Voor dit aanzienlijke voordeel is vereist dat het bewijs als attributengrammatica uit te drukken is, wat het geval is wanneer de semantiek *syntax-gestuurd* is.

**Inferentie.** Voor programmeertalen met een complexe (statische) semantiek is de structuur van het bewijs niet gelijk aan de AST. Tenslotte, om vrijheid in de invulling van het bewijs te hebben, dienen delen van het bewijs van de broncode afleidbaar te zijn, maar niet door de structuur ervan te worden bepaald. Daarvoor bestaan een aantal gangbare algoritmen, zoals het berekenen van een dekpunt van een stelsel van randvoorwaarden, en de gedeeltelijke verkenning van een bos van kandidaat-deelbewijzen. Deze algoritmen hebben als eigenschap dat de attributen wederzijds afhankelijk zijn van *tussentoestanden* van het bewijs. Om bijvoorbeeld een kandidaat te selecteren is het nodig om eigenschappen ervan in te zien. In een attributengrammatica zijn attributen gedefinieerd in termen van het uiteindelijke bewijs, waardoor het lastig is om dergelijke algoritmen met een AG te beschrijven.

In dit proefschrift richten we ons op *geordende* attributengrammatica's, en breiden deze uit met de mogelijkheid om tussentoestanden te inspecteren en te manipuleren. In een geordende AG kan de berekening van de attributen als een eindige sequentie van toestandsveranderingen beschreven worden. Deze beschrijving maakt het mogelijk om over deelbewijzen in een gegeven toestand te redeneren, berekeningsstrategieën te specificeren, en attributen die in deze toestand beschikbaar zijn te inspecteren. Daarvoor schrijven we geen AGs voor de abstracte syntax van de taal, maar AGs voor de abstracte syntax van de bewijsregels van de semantiek.

**Uitbreidingen.** In hoofdstuk 3 introduceren we notatie om een sequentie van *visits* voor een nonterminal te specificeren. Een *visit* is een eenheid van evaluatie voor een knoop in de (bewijs)boom. Ieder attribuut is gerelateerd aan een *visit*. De attributen van een vorige *visit* zijn beschikbaar in een opvolgende *visit*. Bovendien kunnen berekeningen voor attributen aan specifieke *visits* toegekend worden om af te dwingen dat berekeningen in een vaste volgorde plaatsvinden. Deze uitbreiding maakt het mogelijk om monadische operaties met AGs te combineren.

In hoofdstuk 5 laten we zien hoe we berekeningsstrategieën aan *visits* koppelen. Door het voorwaardelijk herhalen van een *visit* aan een knoop kan een dekpunt berekend worden. Met *clauses* kunnen voorwaardelijke berekeningen van attributen en kinderen van de knoop gespecificeerd worden, zodat het mogelijk is om deelbewijzen te verkennen. Ook kunnen knopen ontkoppeld worden en op een andere locatie in de boom weer aangekoppeld worden,

## Samenvatting

waarbij we statische garanties geven over de toestand van dergelijke verplaatsbare knopen. Zo kunnen berekeningen die afhangen van bewijzen die nog niet voltooid zijn uitgesteld worden tot deze bewijzen beschikbaar komen. Hiermee kunnen we *constraints* representeren.

In hoofdstuk 6 gaan we een stapje verder dan visits. Tussenresultaten die tijdens de uitvoering van een visit beschikbaar komen kunnen met technieken uit dit hoofdstuk stapsgewijs geïnspecteerd worden. Door om de beurt kandidaat-knopen een stap te laten zetten, kunnen de bewijzen gelijktijdig verkend worden, zonder de bewijzen van te voren al helemaal op te bouwen.

In hoofdstuk 7 presenteren we AGs met *afhankelijke typen*. Dit zijn AGs waarin het type van een attribuut mag verwijzen naar de waarden van andere attributen. Deze uitbreiding maakt het mogelijk om invarianten op attributen te specificeren en bewijzen ervoor uit te drukken. Voor deze uitbreiding maken we gretig gebruik van de mogelijkheden die door voorgaande hoofdstukken besproken zijn.

De uitbreiding vormen een conservatieve uitbreiding van AGs. De mate van abstractie, zoals deze door AGs aangeboden wordt, blijft behouden. De uitbreidingen maken het mogelijk om eigenschappen van de berekeningsvolgorde te specificeren en te inspecteren, zonder daarbij het automatisch ordenen van attribuutberekeningen te breken. Met de uitbreidingen heeft de programmeur een stel krachtige bouwstenen in handen om compilers mee te implementeren.



# Curriculum Vitae

Adriaan Middelkoop

Born 8 February 1983 in Gorinchem, the Netherlands

## ■ VWO diploma, 2001

Institution: Merewade College Wijdschildlaan, Gorinchem

## ■ Propedeuse Informatica, 2002

Honors: Cum laude

Institution: Universiteit Utrecht

Subject: General track

## ■ M. Sc. Software Technology, 2006

Honors: Cum laude

Institution: Universiteit Utrecht

Specialisation: Programming languages and compilers

Subject: Program analysis, uniqueness typing

## ■ Ph. D. Software Technology, 2011

Institution: Universiteit Utrecht

Subject: Attribute grammars, type inference

## ■ Scientific Programmer, 2010 - 2011

Institution: Universiteit Utrecht

Subject: Program instrumentation

## ■ Postdoc, 2011 - 2012

Institution: Laboratoire d'Informatique de Paris 6, Paris

Subject: Program transformation



## Titles in the IPA Dissertation Series since 2005

- E. Dolstra.** *The Purely Functional Software Deployment Model.* Faculty of Science, UU. 2006-01
- R.J. Corin.** *Analysis Models for Security Protocols.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02
- P.R.A. Verbaan.** *The Computational Complexity of Evolving Systems.* Faculty of Science, UU. 2006-03
- K.L. Man and R.R.H. Schiffelers.** *Formal Specification and Analysis of Hybrid Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04
- M. Kyas.** *Verifying OCL Specifications of UML Models: Tool Support and Compositionality.* Faculty of Mathematics and Natural Sciences, UL. 2006-05
- M. Hendriks.** *Model Checking Timed Automata - Techniques and Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2006-06
- J. Ketema.** *Böhm-Like Trees for Rewriting.* Faculty of Sciences, VUA. 2006-07
- C.-B. Breunesse.** *On JML: topics in tool-assisted verification of JML programs.* Faculty of Science, Mathematics and Computer Science, RU. 2006-08
- B. Markvoort.** *Towards Hybrid Molecular Simulations.* Faculty of Biomedical Engineering, TU/e. 2006-09
- S.G.R. Nijssen.** *Mining Structured Data.* Faculty of Mathematics and Natural Sciences, UL. 2006-10
- G. Russello.** *Separation and Adaptation of Concerns in a Shared Data Space.* Faculty of Mathematics and Computer Science, TU/e. 2006-11
- L. Cheung.** *Reconciling Nondeterministic and Probabilistic Choices.* Faculty of Science, Mathematics and Computer Science, RU. 2006-12
- B. Badban.** *Verification techniques for Extensions of Equality Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13
- A.J. Mooij.** *Constructive formal methods and protocol standardization.* Faculty of Mathematics and Computer Science, TU/e. 2006-14
- T. Krilavicius.** *Hybrid Techniques for Hybrid Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15
- M.E. Warnier.** *Language Based Security for Java and JML.* Faculty of Science, Mathematics and Computer Science, RU. 2006-16
- V. Sundramoorthy.** *At Home In Service Discovery.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17
- B. Gebremichael.** *Expressivity of Timed Automata Models.* Faculty of Science,

Mathematics and Computer Science,  
RU. 2006-18

**L.C.M. van Gool.** *Formalising Interface Specifications.* Faculty of Mathematics and Computer Science, TU/e. 2006-19

**C.J.F. Cremers.** *Scyther - Semantics and Verification of Security Protocols.* Faculty of Mathematics and Computer Science, TU/e. 2006-20

**J.V. Guillen Scholten.** *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition.* Faculty of Mathematics and Natural Sciences, UL. 2006-21

**H.A. de Jong.** *Flexible Heterogeneous Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

**N.K. Kavaldjiev.** *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

**M. van Veelen.** *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of Mathematics and Computing Sciences, RUG. 2007-03

**T.D. Vu.** *Semantics and Applications of Process and Program Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

**L. Brandán Briones.** *Theories for Model-based Testing: Real-time and Coverage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

**I. Loeb.** *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06

**M.W.A. Streppel.** *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07

**N. Trčka.** *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08

**R. Brinkman.** *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

**A. van Weelden.** *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10

**J.A.R. Noppen.** *Imperfect Information in Software Development Processes.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

**R. Boumen.** *Integration and Test plans for Complex Manufacturing Systems.* Faculty of Mechanical Engineering, TU/e. 2007-12

**A.J. Wijs.** *What to do Next?: Analysing and Optimising System Behaviour in Time.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

**C.F.J. Lange.** *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML.* Faculty of Mathematics and Computer Science, TU/e. 2007-14

**T. van der Storm.** *Component-based Configuration, Integration and Delivery.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-15

**B.S. Graaf.** *Model-Driven Evolution of Software Architectures.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

**A.H.J. Mathijssen.** *Logical Calculi for Reasoning with Binding.* Faculty

of Mathematics and Computer Science, TU/e. 2007-17

**D. Jarnikov.** *QoS framework for Video Streaming in Home Networks.* Faculty of Mathematics and Computer Science, TU/e. 2007-18

**M. A. Abam.** *New Data Structures and Algorithms for Mobile Data.* Faculty of Mathematics and Computer Science, TU/e. 2007-19

**W. Pieters.** *La Volonté Machinale: Understanding the Electronic Voting Controversy.* Faculty of Science, Mathematics and Computer Science, RU. 2008-01

**A.L. de Groot.** *Practical Automaton Proofs in PVS.* Faculty of Science, Mathematics and Computer Science, RU. 2008-02

**M. Bruntink.** *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

**A.M. Marin.** *An Integrated System to Manage Crosscutting Concerns in Source Code.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

**N.C.W.M. Braspenning.** *Model-based Integration and Testing of High-tech Multi-disciplinary Systems.* Faculty of Mechanical Engineering, TU/e. 2008-05

**M. Bravenboer.** *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates.* Faculty of Science, UU. 2008-06

**M. Torabi Dashti.** *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

**I.S.M. de Jong.** *Integration and Test Strategies for Complex Manufacturing Machines.* Faculty of Mechanical Engineering, TU/e. 2008-08

**I. Hasuo.** *Tracing Anonymity with Coalgebras.* Faculty of Science, Mathematics and Computer Science, RU. 2008-09

**L.G.W.A. Cleophas.** *Tree Algorithms: Two Taxonomies and a Toolkit.* Faculty of Mathematics and Computer Science, TU/e. 2008-10

**I.S. Zapreev.** *Model Checking Markov Chains: Techniques and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

**M. Farshi.** *A Theoretical and Experimental Study of Geometric Networks.* Faculty of Mathematics and Computer Science, TU/e. 2008-12

**G. Gulesir.** *Evolvable Behavior Specifications Using Context-Sensitive Wildcards.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

**F.D. Garcia.** *Formal and Computational Cryptography: Protocols, Hashes and Commitments.* Faculty of Science, Mathematics and Computer Science, RU. 2008-14

**P. E. A. Dürr.** *Resource-based Verification for Robust Composition of Aspects.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

**E.M. Bortnik.** *Formal Methods in Support of SMC Design.* Faculty of Mechanical Engineering, TU/e. 2008-16

**R.H. Mak.** *Design and Performance Analysis of Data-Independent Stream Processing Systems.* Faculty of Mathematics and Computer Science, TU/e. 2008-17

**M. van der Horst.** *Scalable Block Processing Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2008-18

**C.M. Gray.** *Algorithms for Fat Objects: Decompositions and Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-19

**J.R. Calamé.** *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

**E. Mumford.** *Drawing Graphs for Cartographic Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-21

**E.H. de Graaf.** *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation.* Faculty of Mathematics and Natural Sciences, UL. 2008-22

**R. Brijder.** *Models of Natural Computation: Gene Assembly and Membrane Systems.* Faculty of Mathematics and Natural Sciences, UL. 2008-23

**A. Koprowski.** *Termination of Rewriting and Its Certification.* Faculty of Mathematics and Computer Science, TU/e. 2008-24

**U. Khadim.** *Process Algebras for Hybrid Systems: Comparison and Development.* Faculty of Mathematics and Computer Science, TU/e. 2008-25

**J. Markovski.** *Real and Stochastic Time in Process Algebras for Performance Evaluation.* Faculty of Mathematics and Computer Science, TU/e. 2008-26

**H. Kastenberg.** *Graph-Based Software Specification and Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

**I.R. Buhan.** *Cryptographic Keys from Noisy Data Theory and Applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

**R.S. Marin-Perianu.** *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

**M.H.G. Verhoef.** *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

**M. de Mol.** *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

**M. Lormans.** *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

**M.P.W.J. van Osch.** *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

**H. Sozer.** *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

**M.J. van Weerdenburg.** *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

**H.H. Hansen.** *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

**A. Mesbah.** *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

- A.L. Rodriguez Yakushev.** *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9
- K.R. Olmos Joffré.** *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10
- J.A.G.M. van den Berg.** *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11
- M.G. Khatib.** *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12
- S.G.M. Cornelissen.** *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13
- D. Bolzoni.** *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14
- H.L. Jonker.** *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15
- M.R. Czenko.** *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16
- T. Chen.** *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17
- C. Kaliszyk.** *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18
- R.S.S. O'Connor.** *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19
- B. Ploeger.** *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20
- T. Han.** *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21
- R. Li.** *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22
- J.H.P. Kwisthout.** *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23
- T.K. Cocx.** *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24
- A.I. Baars.** *Embedded Compilers.* Faculty of Science, UU. 2009-25
- M.A.C. Dekker.** *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26
- J.F.J. Laros.** *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27
- C.J. Boogerd.** *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01
- M.R. Neuhäuser.** *Model Checking Non-deterministic and Randomly Timed Systems.*

Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

**J. Endrullis.** *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

**T. Staijen.** *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

**Y. Wang.** *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

**J.K. Berendsen.** *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06

**A. Nugroho.** *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07

**A. Silva.** *Kleene Coalgebra.* Faculty of Science, Mathematics and Computer Science, RU. 2010-08

**J.S. de Bruin.** *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications.* Faculty of Mathematics and Natural Sciences, UL. 2010-09

**D. Costa.** *Formal Models for Component Connectors.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

**M.M. Jaghoori.** *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services.* Faculty of Mathematics and Natural Sciences, UL. 2010-11

**R. Bakhshi.** *Gossiping Models: Formal Analysis of Epidemic Protocols.* Faculty of Sciences, Department of Computer Science, VUA. 2011-01

**B.J. Arnoldus.** *An Illumination of the Template Enigma: Software Code Generation with Templates.* Faculty of Mathematics and Computer Science, TU/e. 2011-02

**E. Zambon.** *Towards Optimal IT Availability Planning: Methods and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

**L. Astefanoaei.** *An Executable Theory of Multi-Agent Systems Refinement.* Faculty of Mathematics and Natural Sciences, UL. 2011-04

**J. Proença.** *Synchronous coordination of distributed components.* Faculty of Mathematics and Natural Sciences, UL. 2011-05

**A. Morali.** *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

**M. van der Bijl.** *On changing models in Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

**C. Krause.** *Reconfigurable Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-08

**M.E. Andrés.** *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2011-09

**M. Atif.** *Formal Modeling and Verification of Distributed Failure Detectors.* Faculty of Mathematics and Computer Science, TU/e. 2011-10

**P.J.A. van Tilburg.** *From Computability to Executability – A process-theoretic view on automata theory.* Faculty of Mathematics and Computer Science, TU/e. 2011-11



- Z. Protic.** *Configuration management for models: Generic methods for model comparison and model co-evolution.* Faculty of Mathematics and Computer Science, TU/e. 2011-12
- S. Georgievska.** *Probability and Hiding in Concurrent Processes.* Faculty of Mathematics and Computer Science, TU/e. 2011-13
- S. Malakuti.** *Event Composition Model: Achieving Naturalness in Runtime Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14
- M. Raffelsieper.** *Cell Libraries and Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-15
- C.P. Tsirogiannis.** *Analysis of Flow and Visibility on Triangulated Terrains.* Faculty of Mathematics and Computer Science, TU/e. 2011-16
- Y.-J. Moon.** *Stochastic Models for Quality of Service of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-17
- R. Middelkoop.** *Capturing and Exploiting Abstract Views of States in OO Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-18
- M.F. van Amstel.** *Assessing and Improving the Quality of Model Transformations.* Faculty of Mathematics and Computer Science, TU/e. 2011-19
- A.N. Tamalet.** *Towards Correct Programs in Practice.* Faculty of Science, Mathematics and Computer Science, RU. 2011-20
- H.J.S. Basten.** *Ambiguity Detection for Programming Language Grammars.* Faculty of Science, UvA. 2011-21
- M. Izadi.** *Model Checking of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-22
- L.C.L. Kats.** *Building Blocks for Language Workbenches.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23
- S. Kemper.** *Modelling and Analysis of Real-Time Coordination Patterns.* Faculty of Mathematics and Natural Sciences, UL. 2011-24
- A. Khosravi.** *Optimal Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2012-01
- A. Middelkoop.** *Inference of Program Properties with Attribute Grammars, Revisited.* Faculty of Science, UU. 2012-02

