Thesis for the Degree of Doctor of Engineering

Programming Language Techniques for Natural Language Applications

Björn Bringert

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Göteborg
Sweden

Gothenburg, October 2008

Programming Language Techniques for Natural Language Applications
Björn Bringert
ISBN 978-91-628-7618-0

Technical report 48D
Department of Computer Science and Engineering
Language Technology Research Group

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31 772 1000

Printed at Chalmers, Gothenburg, Sweden, 2008

Abstract

It is easy to imagine machines that can communicate in natural language. Constructing such machines is more difficult. The aim of this thesis is to demonstrate how declarative grammar formalisms that distinguish between abstract and concrete syntax make it easier to develop natural language applications.

We describe how the type-theorectical grammar formalism Grammatical Framework (GF) can be used as a high-level language for natural language applications. By taking advantage of techniques from the field of programming language implementation, we can use GF grammars to perform portable and efficient parsing and linearization, generate speech recognition language models, implement multimodal fusion and fission, generate support code for abstract syntax transformations, generate dialogue managers, and implement speech translators and web-based syntax-aware editors.

By generating application components from a declarative grammar, we can reduce duplicated work, ensure consistency, make it easier to build multilingual systems, improve linguistic quality, enable re-use across system domains, and make systems more portable.

Table of Contents

Acknowledgments

First and foremost, I want to thank my supervisor and co-author Aarne Ranta for his friendship and constant support. Robin Cooper, Peter Ljunglöf, Krasimir Angelov and Moisés Salvador Meza Moreno have also helped me write the articles in this thesis, thank you! Together with Aarne and Robin, Bengt Nordström and Koen Claessen make up my PhD advisory committee, who have kept me on track and going forward. The other members of the Chalmers Language Technology Group, the Centre for Language Technology, and the TALK project, including Harald Hammarström, Markus Forsberg, Håkan Burden, Kristofer Johannisson, Janna Khegai, K. V. S. Prasad, Muhammad Humayoun, David Hjelm, Staffan Larsson, Stina Ericsson, Rebecca Jonson, Jessica Villing, Ann-Charlotte Forslund, Andreas Wallentin, Mikael Sandin, Ali El Dada, Hans-Joachim Daniels, Lars Borin, and Torbjörn Lager have all contributed to my work in various ways and helped make this a creative research environment. Rolf Carlson at KTH gave me access to speech data and kindly agreed to serve as discussion leader at my licentiate seminar. Nuance Communications Inc., OptimSys s.r.o., Opera Software ASA, and ROBO Design have provided me with software licenses and technical support. I have received data, suggestions and bug reports from many people, including Glòria Casanellas at Maths for More, Jordi Saludes at Universitat Politècnica de Catalunya, Wanjiku Ng'ang'a at the University of Helsinki, Oliver Lemon and Xingkun Liu at the University of Edinburgh, Manny Rayner at the University of Geneva, Nadine Perera at BMW Group Research and Technology, Jochen Steigner at DFKI, Joel Reymont at Wager Labs SA, Steve Legrand at the University of Jyväskylä, Sara Stymne at Linköping University, and Howard Bartel at RadiSys Corporation. There is an even wider circle of people with whom I have had the pleasure to interact since I came to the Computer Science and Engineering department as an undergraduate. Thanks are due to everyone at this outstanding department. Wojciech Mostowski and Andres Löh have provided valuable help with the LaTeX and `lhs2TeX` wrangling required for typesetting this thesis. Angela, thank you for your love and care. Thank you for bringing me happiness and for helping me grow. My son Tor, seeing your face every day brings me incredible joy. My parents Ebba and Gösta and my brother Klas, thank you for the loving and intellectually stimulating home that I grew up in. In order to help the reader continue past this page, there are further acknowledgments in some of the included papers.

Björn Bringert

Gothenburg
October 2008

Introduction

This thesis shows how Grammatical Framework (GF, Ranta 2004) grammars can be used to simplify development of interactive natural language applications. This chapter provides some background on natural language applications and GF, and introduces the seven articles that make up the bulk of the thesis. The articles describe how GF grammars can be used for speech recognition grammars, multimodality, semantic transfer, dialogue management, portable parsing and generation, syntax-aware text editing, and speech translation.

The articles are presented in the context of interactive speech applications since this is an area where the ideas can be used together. There are also other applications of these results, which is perhaps most readily apparent in Paper VII, where most of the examples are from programming language implementation rather than natural language processing.

1 Interactive Natural Language Applications

What do we mean by interactive natural language applications? By *interactive*, we mean that the system gets input from a user, and delivers timely output as a result of user input. There is some relation between the input and output. A single task may be accomplished using one or more input/output interactions. Most of the examples in this thesis concern *speech applications*, that is, applications which get speech input from the user, and deliver speech output to the user. In the case of a speech translator, one user produces the input, and another receives the output. An interactive natural language application may also be *multimodal*, that is, it may use multiple modes of communication, or *modalities*. Gestures and drawings are possible examples of modalities other than speech. Both the user input and the system output can be multimodal. Systems which are not multimodal are called *unimodal*.

Interactive speech applications have long been a staple of science fiction. Figures 1 and 2 illustrate two such applications: a *speech translator*, and a *dialogue system*. Interactive speech applications are already in limited commercial use. Examples of such applications include phone-based travel reservation systems and speech-enabled phrase books. However, interactive speech applications have yet to have a significant impact on everyday life. There are three major problems with current interactive speech applications:

1. They are not natural enough.
2. They are not usable enough.

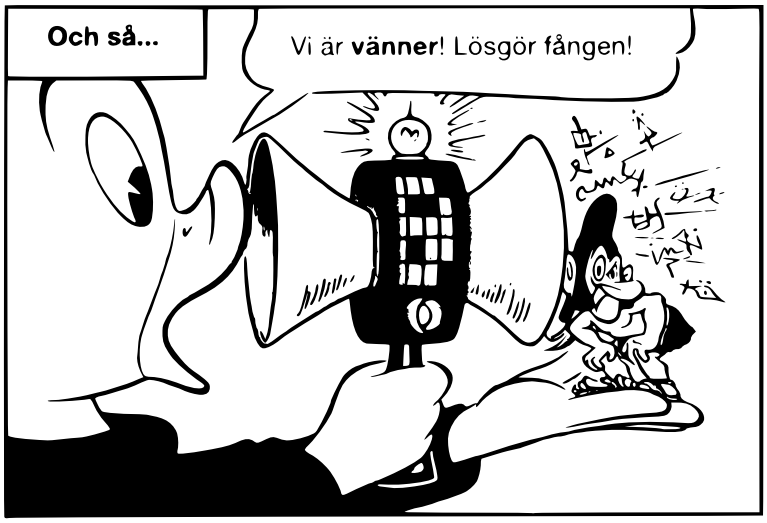

Figure 1. A speech translator, from the Uncle Scrooge story "Planet-planering" (English title "Scrooge's Space Show") (Branca et al. 1987). Louie says "We are friends! Release the prisoner!", though in the original English version he says "Our uncle is a mega-merchant come to trade with you guys!". ©Disney. Located with help from Sivebæk, Willot and Jensen (2007).

Figure 2. A dialogue system, from the Uncle Scrooge story "Operation Hajön" (English title "Operation Gootchy-goo") (Strobl and Steere 1985). Uncle Scrooge says: "Stop babbling about the weather! I want to know if everything is proceeding according to plan!". Smedly, the computer, responds "Right! Well, let me see. . . ", quite like a GoDiS (Larsson 2002) dialogue system would. ©Walt Disney Productions. Located with help from Sivebæk, Willot and Jensen (2007).

3. They are not cheap enough.

According to Pieraccini (2005), academic dialogue system research is largely focused on the problem of naturalness, whereas industrial dialogue system development is more concerned with usability. Waibel (2004) considers high development cost and limited domains to be the major problems in speech translation research.

There are many applications where the current state of the art means that it is *not possible* to build systems that are natural or usable enough. However, there are also many applications which could benefit from use of even the current state of interactive speech technology, but where it is *not economically viable*, because of the high cost of implementing the systems.

1.1 Problems

The problems of naturalness, usability and cost are large and complex. This thesis deals with the following sub-problems:

Duplicated work In current practice, multiple components are developed independently, with much duplicated effort. For example, speech recognition grammars, semantic interpretation, and output generation all need to take into account the linguistic and domain-specific coverage of the system.

Consistency It is difficult to modify a system which uses multiple hand-written components. The problem is multiplied when the system is multilingual. The developer then has to modify each of the components, such as speech recognition grammars and semantic interpretation, manually for each language. A simple change may require touching many parts of the system, and there are no automatic consistency checks.

Localization With hand-written components, it is about as difficult to add support for a new language as it is to write the grammar, semantic interpretation, and generation components for the first language.

Linguistic quality Because of the lack of powerful language description tools, achieving high syntactic and morphological quality of the system output and the input language models can be costly. This is more pronounced for languages with a richer morphology than English, since current methods are often developed with English in mind.

Domain portability Components implemented for a given application domain can often not be easily reused in other domains.

Platform portability Systems implemented for a given platform (speech recognizer, operating system, programming language, etc.) can often not be used on other platforms.

2 This work

Our aim is to make the construction of interactive natural language applications easier by compiling high-level specifications to the low-level code used in the running system. GF is "the working programmer's grammar formalism". In this spirit, the approach that we have taken is to use techniques borrowed from programming language implementation to automatically generate system components from grammars.

In the early days of computer programming, programs were written in machine code or assembly languages, very low-level languages which give the programmer full control, but make programs hard to write, limited to a single machine architecture, and difficult to maintain. Today, programs are written in high-level languages which give less control, but make programs easier to write, portable across different machines and operating systems, and easier to maintain. Programs written in high-level languages are compiled to code in low-level languages, which can be run by machines.

The approach to development of interactive natural language applications which we describe here is *grammar-based*, since we use high-level grammars to define major parts of the functionality. Several different components used in interactive natural language applications can be generated automatically from the grammars. The systems which we generate are *rule-based*, rather than *statistical*. In an experiment by Rayner et al. (2005a), a rule-based speech understanding system was found to outperform a statistical one, and the advantage of the rule-based system increased with the users' familiarity with the system.

In our description of the components which we generate, we consider interactive natural language applications which can be implemented as pipelines. The system receives input, which is processed step by step, and in the end output is produced. A multimodal dialogue system may have components such as speech recognition, multimodal fusion, parsing, dialogue management, domain resources, output generation, linearization, multimodal fission, and speech synthesis. Figure 3 shows a schematic view of such a system. In a speech translator, the dialogue management and domain resources may be replaced by a semantic transfer component, as shown in Figure 4. Larger systems, such as the Spoken Language Translator (Rayner et al. 2000), are more complex with more components and an architecture which is not a simple pipeline. The individual components that we generate can be used in more complex architectures, as has been done in experimental dialogue systems (Ericsson et al. 2006) which use the GoDiS (Larsson 2002) implementation of issue-based dialogue management.

2.1 Advantages

This work addresses the problems listed in Section 1.1 in the following ways:

Duplicated work The duplicated work involved in developing multiple components is avoided by generating all the components from a single declarative source, a GF grammar.

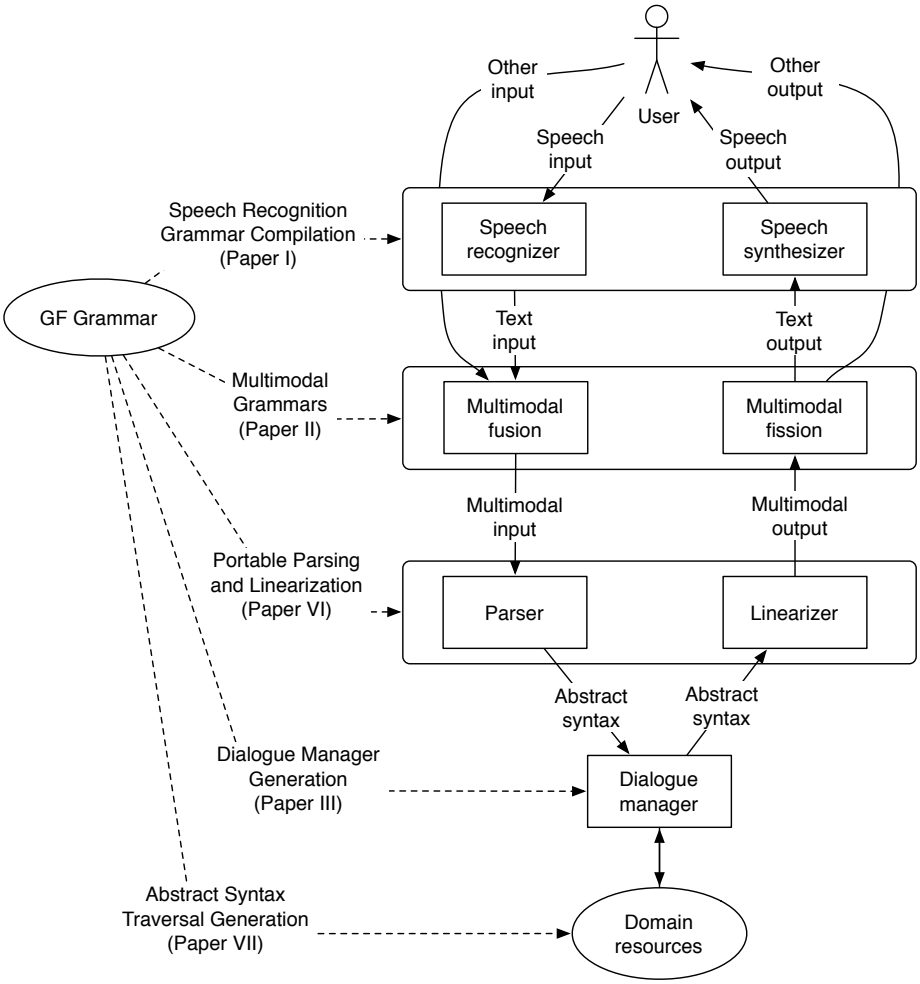

Figure 3. Architecture of a grammar-based multimodal dialogue system. In a unimodal system, there would be no multimodal fission and fusion components.

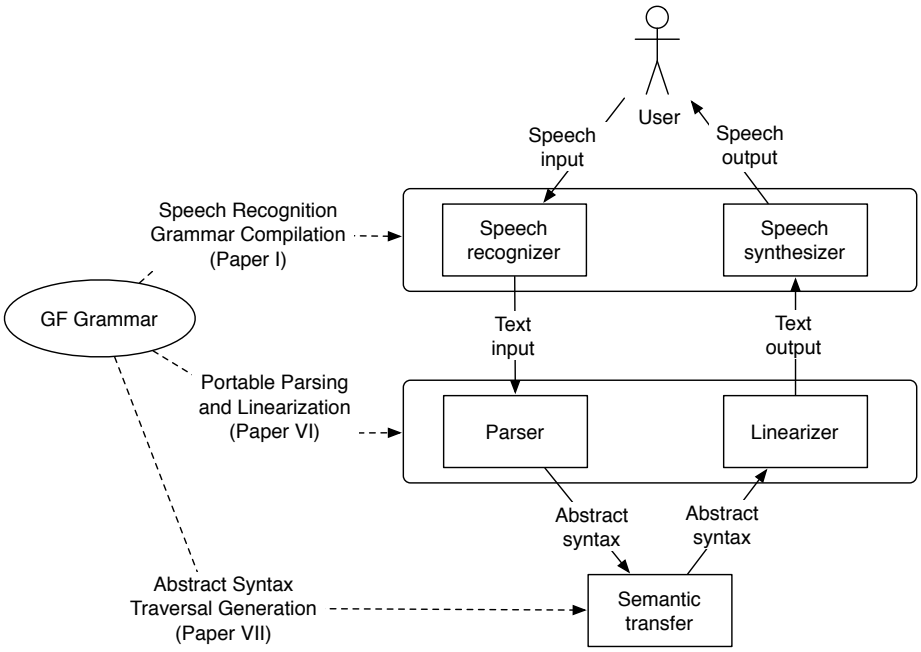

Figure 4. Architecture of a grammar-based speech translator. Compared to Figure 3, there is no multimodality, and the dialogue manager and domain resources have been replaced by a semantic transfer engine.

Consistency The strong typing of the GF language enforces consistency between the abstract syntax and the concrete syntaxes for each language. This makes it easier to keep the semantics and the implementations for different languages in sync.

Localization GF's support for multilingual grammars and the common interface implemented by all grammars in the GF resource grammar library makes it easy to translate a system to a new language.

Linguistic quality GF's powerful constructs and the multilingual resource grammar library allows for high morphological and syntactic quality at a low cost.

Domain portability A large portion of the grammar implementation effort is in the resource grammar library. This library can be re-used in multiple applications, instead of being re-implemented every time.

Platform portability In our approach, a GF grammar is used as the canonical representation which the developer works with, and components in any of a number of formats can be generated automatically from this representation.

2.2 Limitations

The goal is not mainly to allow more sophisticated applications, but rather to reduce the development cost of medium complexity applications. Just like high-level programming languages take away some of the control that the assembly language programmer has, generating system components from grammars places some limits on how systems can be implemented. These limits of course depend on how sophisticated the generation is.

Taken together, the components that we generate fit best in systems with pipeline architectures like the one shown in Figure 3. However, the individual components could also be used as parts of systems with other architectures. For example, a hybrid system could use our components to attempt a deep analysis, and fall back to a separate statistical language model and shallow syntactic analysis when that fails.

3 Grammatical Framework

We use Grammatical Framework (GF, Ranta 2004) as the source language for our component generation. This section gives a short introduction to GF, with a small example grammar for a dialogue system.

GF is a type theoretic grammar formalism based on Martin-Löf's (1984) type theory. GF makes a distinction between *abstract syntax* and *concrete syntax*, corresponding to Curry's (1961) division of grammar into *tectogrammar* and *phenogrammar*. The abstract syntax declares *what* can be said in the language. Figure 5 shows an example of a simple abstract syntax module. The concrete syntax describes *how* to say it, by associating a concrete representation with

```
abstract Agreement = {
  cat S; NP; VP;
  fun pred : NP → VP → S;
      john : NP;
      walk : VP;
}
```

Figure 5. An abstract syntax module. The only possible tree in the S category is *pred john walk*.

```
concrete AgreementEng of Agreement = {
  param Num = Sg | Pl;
  lincat S  = Str;
         NP = {s : Str; n : Num};
         VP = Num ⇒ Str;
  lin pred np vp = np.s ++ vp ! np.n;
      john = {s = "John"; n = Sg};
      walk = table {Sg ⇒ "walks"; Pl ⇒ "walk"};
}
```

Figure 6. A concrete syntax for the abstract syntax in Figure 5. The linearization type of the NP category includes a number field, and the linearization type of the VP category is an inflection table that takes a number argument. In the linearization of *pred*, the form of the verb phrase is selected according to the number of the noun phrase. In this concrete syntax, the abstract syntax tree *pred john walk* is linearized to the string "John walks".

each construct in the abstract syntax. In the simplest case, this concrete representation is a single string. Records, tables and enumerations can be used to implement more complex representations, for example with number agreement between nouns and verbs. The process of generating a concrete syntax term from an abstract syntax tree is called *linearization*. Figure 6 shows an example of a concrete syntax module.

One of GF's strong points is multilinguality. The division of grammar into abstract and concrete syntax means that it is possible to have multiple concrete syntaxes for one abstract syntax. We call this a *multilingual grammar*. In order to avoid re-implementing the domain-independent linguistic details of a language for each new application grammar, the GF Resource Grammar Library (Ranta 2008) has been created. It implements the morphological and syntactic details of a number of languages, and presents a language-independent API to the application grammar writer. This significantly reduces the effort involved in translating grammars (Perera and Ranta 2007).

```
abstract Pizza = {
  flags startcat = Input;
  cat Input;
      Order;
      Number;
      Size;
      Topping;
      [Topping]{1};
  fun order        : Order → Input;
      pizza        : Number → Size → [Topping] → Order;
      one, two     : Number;
      small, large : Size;
      cheese, ham  : Topping;
  cat Output;
  fun price : Order → Number → Output;
}
```

Figure 7. A GF abstract syntax module.

3.1 An Example Application Grammar

As an example of a GF grammar for an interactive natural language application, Figure 7 shows the abstract syntax for a small pizza ordering dialogue system. The Input category contains user input, such as "one large pizza with ham and cheese", which corresponds to the abstract syntax tree *order* (*pizza one large* [*ham*, *cheese*]). The Output category describes system output, for example "one large pizza with ham and cheese costs two euros", for the abstract syntax tree *price* (*pizza one large* [*ham*, *cheese*]) *two*.

Figure 8 shows a parametrized concrete syntax module (Ranta 2007) which uses the language-independent part of the Resource Library API (Ranta 2008), and a domain-specific lexicon interface. Each function *mkX* constructs a term in the resource grammar category X. For example, the linearization category of the Order category is the resource grammar category NP. This means that an order is represented by a noun phrase in this concrete syntax. The *mkNP* function is overloaded, and the version of it that is used in the linearization of *pizza* takes two arguments, one of type Numeral (number word, the linearization category of Number) and one of type CN (common noun, here constructed from another common noun and an adverbial phrase). In the linearization of *cheese*, a version of *mkNP* is used that takes a simple common noun (N) and returns a mass expression. The linearization of *ConsTopping*, one of the two constructors in the [Topping] category, uses a third version of *mkNP* that takes a conjunction and two noun phrases as arguments.

The parametrized concrete syntax is instantiated for English as shown in Figure 9. One noteworthy feature is that the lexicon entry for *large_A* uses

```
incomplete concrete PizzaI of Pizza = open Syntax, PizzaLex in {
  lincat Input      = Utt;
         Order      = NP;
         Number     = Numeral;
         Size       = AP;
         Topping    = NP;
         [Topping]  = NP;
  lin order o              = mkUtt o;
      pizza n s ts         = mkNP n (mkCN (mkCN s pizza_N)
                                  (mkAdv with_Prep ts));
      one                  = n1_Numeral;
      two                  = n2_Numeral;
      small                = mkAP small_A;
      large                = mkAP large_A;
      cheese               = mkNP cheese_N;
      ham                  = mkNP ham_N;
      BaseTopping t        = t;
      ConsTopping t ts     = mkNP and_Conj t ts;
  lincat Output = Utt;
  lin price o p = mkUtt (mkCl o cost_V2 (mkNP p euro_N));
}

interface PizzaLex = open Cat in {
  oper pizza_N  : N;
       small_A  : A;
       large_A  : A;
       cheese_N : N;
       ham_N    : N;
       cost_V2  : V2;
       euro_N   : N;
}
```

Figure 8. Parametrized concrete syntax for the abstract syntax in Figure 7.

```
instance PizzaLexEng of PizzaLex = open CatEng, ParadigmsEng in {
  oper pizza_N  = mkN "pizza";
       small_A  = mkA "small";
       large_A  = mkA "large" | mkA "big";
       cheese_N = mkN "cheese";
       ham_N    = mkN "ham";
       cost_V2  = mkV2 (mkV "cost");
       euro_N   = mkN "euro" "euros";
}

concrete PizzaEng of Pizza = PizzaI with (Syntax    = SyntaxEng),
                                         (PizzaLex = PizzaLexEng);
```

Figure 9. English concrete syntax for the abstract syntax in Figure 7, using the parametrized concrete syntax in Figure 8.

variants, to allow alternative linearizations. This is used extensively in realistic dialogue system grammars, to handle variation in how input can be expressed without complicating the abstract syntax. Figure 10 shows how the parametrized concrete syntax can be instantiated to create a German concrete syntax.

4 Paper I: Speech Recognition Grammar Compilation in Grammatical Framework

Speech recognizers use *speech recognition grammars* (also known as *language models*) to limit the input language in order to achieve acceptable recognition performance. In the paper "Speech Recognition Grammar Compilation in Grammatical Framework", we show how speech recognition grammars in several commonly used context-free and finite-state formalisms can be generated from GF grammars. We also describe generation of semantic interpretation code which can be embedded in speech recognition grammars.

4.1 An Example

For the Input category in the example grammar in Figure 7 and Figure 9, we can generate the finite-state model shown in Figure 11. Finite-state models such as this one are used to guide the HTK speech recognizer.

```
instance PizzaLexGer of PizzaLex = open CatGer, ParadigmsGer in {
  flags coding = utf8;
  oper pizza_N  = mkN "Pizza" "Pizzas" feminine;
       small_A  = mkA "klein";
       large_A  = mkA "groß" "größer" "größte";
       cheese_N = mkN "Käse" "Käse" masculine;
       ham_N    = mkN "Schinken";
       cost_V2  = mkV2 (mkV "kostet");
       euro_N   = mkN "Euro" "Euros" masculine;
}

concrete PizzaGer of Pizza = PizzaI with (Syntax    = SyntaxGer),
                                         (PizzaLex = PizzaLexGer);
```

Figure 10. German concrete syntax for the abstract syntax in Figure 7.

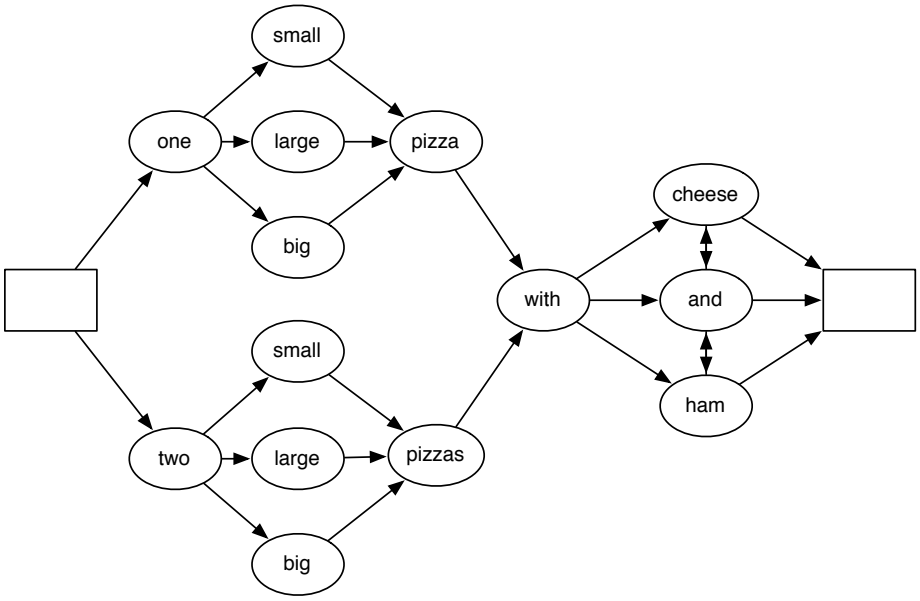

Figure 11. Finite-state language model generated from the English concrete syntax in Figure 9.

4.2 Contribution

I wrote the paper myself, and I implemented the various grammar translations it describes.

4.3 Publication

This paper was published in the *Proceedings of the ACL 2007 Workshop on Grammar-Based Approaches to Spoken Language Processing*, Prague, Czech Republic, June 29, 2007, pages 1–8, by the Association for Computational Linguistics. This thesis includes an updated version of the paper, which describes a new PMCFG-based conversion algorithm and a new non-recursive SRGS back-end.

5 Paper II: Multimodal Dialogue System Grammars

The paper "Multimodal Dialogue System Grammars" describes how GF grammars can be used to handle multimodality, that is, information presented using multiple modes of communication. Multimodal systems can for example combine speech and pointing gestures for input, or speech and graphics for output. *Multimodal fusion*, the integration of information from multiple modalities into a single semantic representation, and *multimodal fission*, the conversion of a single semantic representation into information in multiple modalities, are handled by using GF's facilities for parsing and linearization, respectively.

5.1 An Example

We can extend the example grammar from Section 3 to make a multimodal application. For example, we can write a new concrete syntax which generates drawing instructions instead of natural language utterances. We refer to this as *parallel multimodality*, since the complete information is presented independently in each of the modalities. Figure 12 shows a concrete syntax which generates instructions in a simple drawing language. This can be used to draw graphical representations of pizza orders. Figure 13 shows the graphical representation of the order "two large pizzas with ham and cheese". The abstract syntax representation of this order is *order* (*pizza two large* [*ham*, *cheese*]) and from this, the PizzaDraw concrete syntax generates the drawing instructions: *scale* (1.0, *replicate* (2, *above* (*above* (*image* ("ham"), *image* ("cheese")), *image* ("pizza")))).

Another possible multimodal extension is to allow spoken pizza orders to contain non-speech parts. For example, we can allow the user to say "I want a large pizza with cheese and that", accompanied by a click on a picture of some topping. We refer to this as *integrated multimodality*, and implement it by using one record field per modality in the concrete syntax.

```
concrete PizzaDraw of Pizza = {
  lin order o          = o;
      pizza n s ts     = {s = call2 "scale" s.s (call2 "replicate" n.s
                                (call2 "above" ts.s (image "pizza")))};
      one              = {s = "1"};
      two              = {s = "2"};
      small            = {s = "0.5"};
      large            = {s = "1.0"};
      cheese           = {s = image "cheese"};
      ham              = {s = image "ham"};
      BaseTopping t    = {s = t.s};
      ConsTopping t ts = {s = call2 "above" t.s ts.s};
  oper call0  : Str → Str = λf → f ++ "(" ++ ")";
       call1  : Str → Str → Str = λf, x → f ++ "(" ++ x ++ ")";
       call2  : Str → Str → Str → Str = λf, x, y →
                                          f ++ "(" ++ x ++ "," ++ y ++ ")";
       image : Str → Str = λx → call1 "image" ("\"" + x + "\"");
}
```

Figure 12. A concrete syntax which generates drawing instructions from pizza orders.

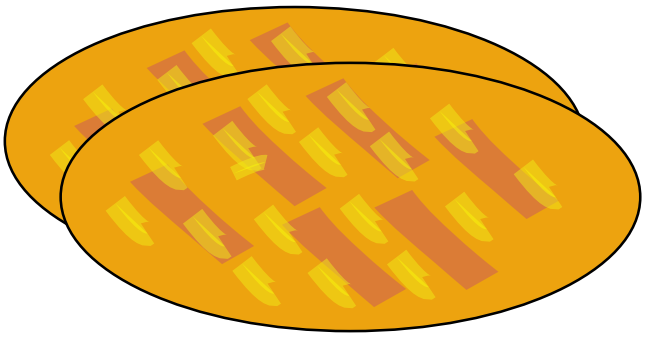

Figure 13. A graphical representation of the pizza order *order* (*pizza two large* [*ham*, *cheese*]), drawn using instructions generated by the concrete syntax in Figure 12.

5.2 Contribution

I designed and implemented the demonstration system, including the grammars, and wrote the sections about the proof-of-concept implementation and related work.

5.3 Publication

This paper was published in the *Proceedings of DIALOR'05, Ninth Workshop on the Semantics and Pragmatics of Dialogue*, Nancy, France, June 9–11, 2005, pages 53–60.

6 Paper III: Rapid Development of Dialogue Systems by Grammar Compilation

The paper "Rapid Development of Dialogue Systems by Grammar Compilation" describes how complete dialogue systems can be generated from GF grammars. It makes use of the speech recognition grammar compiler described in Paper I, and adds two new compilers: one which compiles a GF abstract syntax along with a question for each category to VoiceXML code, and one which compiles GF linearization rules to JavaScript code. The generated VoiceXML code handles the dialogue flow, using a generated speech recognition grammar to get input from the user, and a generated JavaScript linearizer to produce output.

6.1 An Example

If we extend the grammar shown in Figures 7 and 9 with some linearization variants that suppress some parts of the tree, and a question for each category that the system may ask for, we can use it to generate a VoiceXML dialogue system. These changes are shown in Figure 14.

To produce output after the input dialogue has been completed, the system can construct abstract syntax trees and linearize them with the generated JavaScript linearizer. For example, if we add code that calculates the price of an order given an abstract syntax tree for the order, the system could produce the abstract syntax tree *price* (*pizza two large* [*ham*, *cheese*]) *two*, which would be linearized to "two large pizzas with ham and cheese cost two euros". The resulting system allows dialogues such as:

S: What would you like to order?
U: two pizzas
pizza two ? ?
S: What size pizzas do you want?
U: large
pizza two large ?
S: What toppings do you want?
U: ham and cheese

```
printname cat Input      = "What would you like to order?";
              Number     = "How many pizzas do you want?";
              Size       = "What size pizzas do you want?";
              [Topping]  = "What toppings do you want?";
lin pizza n s ts = mkNP n (mkCN sp (mkAdv with_Prep ts) | sp)
    where {sp : CN = mkCN s pizza_N | mkCN pizza_N};
```

Figure 14. Changes to the English concrete syntax in Figure 9 to allow it to be used to generate a VoiceXML dialogue system. GF's **printname** judgement is used for questions for each category that the dialogue system may want input in, and the linearization of the *pizza* function now has several variants that suppress parts of the tree.

pizza two large [*ham*, *cheese*]
S: two large pizzas with ham and cheese cost two euros

Making use of the the ideas from Paper II, we can add a second output modality, as shown in Figure 12. If we add a small interpreter for the generated drawing instructions, we have a multimodal dialogue system, of which a screenshot is shown in Figure 15.

6.2 Contribution

I am the sole author of this paper.

6.3 Publication

This thesis includes an extended version of a short paper published in the *Proceedings of the 8th SIGdial Workshop on Discourse and Dialogue*, Antwerp, Belgium, September 1–2, 2007, pages 223–226, by the Association for Computational Linguistics.

7 Paper IV: Speech Translation with Grammatical Framework

The paper "Speech Translation with Grammatical Framework" explains how the speech recognition grammar compiler described in Paper I and the portable grammar format presented in Paper VI can be used together to build high-precision speech translation systems.

Since the speech recognition grammar compiler supports many speech recognition grammar formats and the portable grammar format has implementations in several programming languages, speech translators built with these components can be used on a number of different platforms.

Figure 15. Multimodal output in a generated XHTML+Voice dialogue system.

7.1 An Example

Say that we want to build a small speech translator that translates pizza orders from German to English. We can compile the German concrete syntax shown in Figure 10 to a speech recognition grammar, and use that to guide an off-the-shelf German speech recognizer. We then use the PGF interpreter to parse the input with the German concrete syntax, and linearize the resulting abstract syntax tree with the English concrete syntax shown in Figure 9. The English text can then be fed to an English speech synthesizer. This system will allow interactions such as the following:

U: zwei große Pizzas mit Schinken und Käse
order (*pizza two large* [*ham*, *cheese*])
S: two large pizzas with ham and cheese

7.2 Contribution

I am the sole author of this paper and the proof-of-concept systems.

7.3 Publication

This short paper was published in *Coling 2008: Proceedings of the workshop on Speech Processing for Safety Critical Translation and Pervasive Applications*, Manchester, UK, August 23, 2008, pages 5–8.

8 Paper V: Interactive Multilingual Web Applications with Grammatical Framework

In the paper "Interactive Multilingual Web Applications with Grammatical Framework", we present a web-based syntax-aware editor based on JavaScript code generated from a GF grammar, and describe how it can be used to build interactive multilingual web applications. In contrast to existing multilingual web applications where content is edited independently in each language, we use a GF abstract syntax as a canonical language-independent content representation. With multiple concrete syntaxes, the content can be linearized to any supported language for viewing. The syntax editor lets users edit the content by abstract syntax manipulation or by parsing strings in any supported language.

8.1 An Example

Figure 16 shows the web-based syntax editor being used to edit a tree in the example pizza order grammar. The current abstract syntax tree is *order* (*pizza two small*?), and the meta-variable of type ListTopping is selected. A possible next step for the user would be to refine the meta-variable with the function $ConsTopping : \mathsf{Topping} \to \mathsf{ListTopping} \to \mathsf{ListTopping}$.

8.2 Contribution

This paper is based on a part of Moisés Salvador Meza Moreno's Master's thesis (Meza Moreno 2008), which I supervised. I suggested the idea of a JavaScript based syntax editor, and helped Moisés with the development and writing. The article is the result of joint writing and editing, based on text from Moisés' thesis.

8.3 Publication

This paper was published in *Advances in Natural Language Processing, 6th International Conference, GoTAL 2008*, Gothenburg, Sweden, August 25–27, 2008, pages 336–347, volume 5221 of Lecture Notes in Computer Science, by Springer.

9 Paper VI: PGF: A Portable Run-Time Format for Type-Theoretical Grammars

The paper "PGF: A Portable Run-Time Format for Type-Theoretical Grammars" describes a portable low-level format which can be used for a number of language processing tasks, including parsing and linearization. The paper also outlines how GF grammars are compiled to PGF, and how PGF grammars can be compiled to other formats. The main goal of PGF is to make it possible to

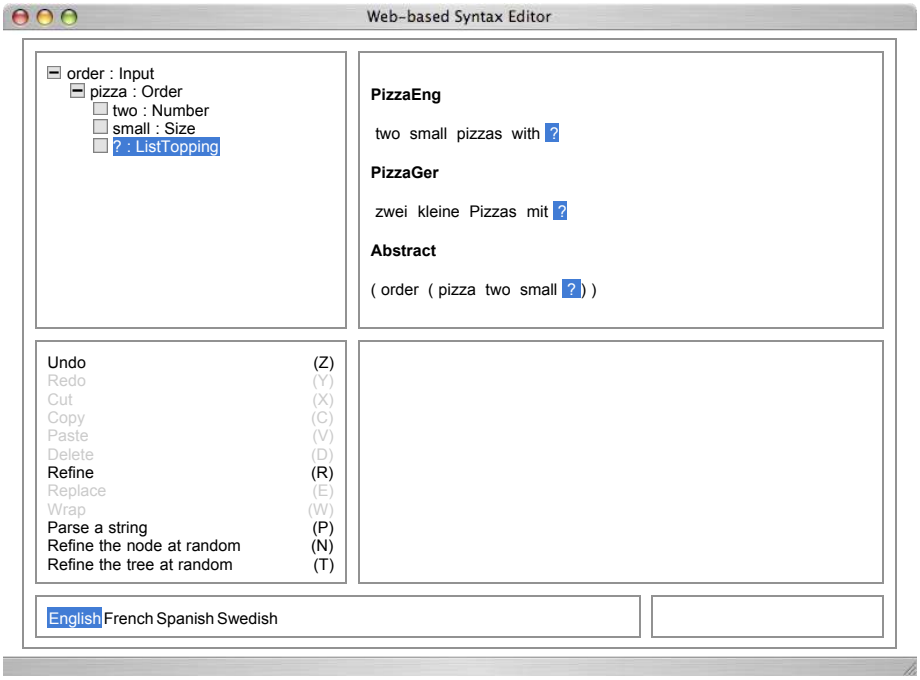

Figure 16. Web-based syntax editor for the grammar in Figures 7, 9, and 10. The top left panel contains the abstract syntax tree which is being edited. The top right panel shows the linearizations in the available concrete syntaxes, with the substrings that correspond to the selected abstract syntax subtree highlighted.

create independent parsing and linearization implementations in any programming language, in order to make it easier to use GF grammars in applications. The new format (PGF, Portable Grammar Format) is much simpler than the GF source language that grammar writers use, but it is still expressive enough to capture all of GF's functionality.

9.1 An Example

We can use the GF grammar compiler to translate the example grammar shown in Figures 7, 9, and 10 to a PGF grammar. We can then load the Haskell PGF interpreter into a Haskell interpreter, and read in the PGF grammar:

PGF > *pgf* ← *readPGF* "Pizza.pgf"

We can now use the grammar for parsing:

PGF > **let** *trees* = *parse pgf* "PizzaEng" "Input" "one large pizza with ham"
PGF > *map showTree trees*
["order (pizza one large (BaseTopping ham))"]

And for linearization:

PGF > *map* (*linearize pgf* "PizzaGer") *trees*
["eine große Pizza mit Schinken"]

9.2 Contribution

This paper is joint work with Krasimir Angelov and Aarne Ranta. I helped design the PGF language, and rewrote the existing compilers that produce speech recognition grammars and VoiceXML to use PGF as the input language. In the paper, I am responsible for the sections Using PGF, and Results and evaluation.

9.3 Publication

This paper has been submitted to the *Journal of Logic, Language and Information*, Special Issue on New Directions in Type-theoretic Grammar.

10 Paper VII: A Pattern for Almost Compositional Functions

The paper "A Pattern for Almost Compositional Functions" describes a method for simplifying a common class of functions over rich tree-like data types, such as abstract syntax trees in compilers or natural language applications. The method uses a type-specific traversal function, which can be automatically generated from the definition of the data type. This method helps reduce the amount of repetitive traversal code in programs which process rich tree structures.

Dialogue managers and semantic transfer engines process abstract syntax trees in various ways. There is a significant set of such transformations that are only concerned with some of the constructs in the often quite rich abstract syntax. This paper describes a way to express such transformations succinctly.

10.1 An Example

Figure 17 shows a small example function which transforms abstract syntax trees. It goes through the tree, removing any duplicate toppings in pizza orders. For example, the tree *order* (*pizza two large* [*ham*, *cheese*, *ham*]) ("two large pizzas with ham and cheese and ham") is transformed to *order* (*pizza two large* [*ham*, *cheese*]) ("two large pizzas with ham and cheese"). Note that in Figure 17, there is no case for *order*. Instead, this case is handled by the *composOp* function, which applies the given function everywhere in a tree. In this small example, not a lot of code is saved, but in a realistic system with many abstract syntax constructors, the code savings can be significant.

```
uniqueToppings :: ∀a.Tree a → Tree a
uniqueToppings t = case t of
                     pizza n s ts → pizza n s (nub ts)
                     _            → composOp uniqueToppings t
```

Figure 17. Haskell-like pseudo-code for an abstract syntax transformation function which removes duplicate toppings from trees in the abstract syntax in Figure 7. The *nub* function removes duplicate elements from a list.

10.2 Contribution

Aarne Ranta used a first version of *composOp* in the GF implementation. He then generalized this to constructive type theory and wrote a first version of the paper, describing the single data-type Haskell versions of *composOp* and *composM*, and a type family version of *composOp* in Agda.

I extended this first version to the full paper included here. Aarne's original paper represents about one fifth of the text of the final version. My contributions include support for *composOp* over type families in extended Haskell, the general *compos* function, the library of functions which use *compos*, the Java Visitor version of the pattern, the description of the relationship to applicative functors, including some identities with proofs, and descriptions of the relationships to generic programming in Haskell, tree types in type theory, and other related work.

10.3 Publication

This paper was first published in *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, Portland, Oregon, September 18–20, 2006, pages 216–226, by ACM Press. The proceedings were also published as *ACM SIGPLAN Notices*, volume 41, issue 9. An extended version has been accepted for publication in the *Journal of Functional Programming* (JFP). This thesis includes a pre-print version of the JFP paper.

11 Related Work

11.1 GF in Interactive Speech Applications

Jonson (2006, 2007) shows how to generate statistical language models for dialogue systems from GF grammars, instead of the grammar-based models that are described in Paper I. This was found to improve the recognition performance on out-of-grammar utterances significantly, without any substantial negative impact on in-grammar performance.

Ranta and Cooper (2004) show how a proof editor for type theory can be used to implement information-seeking dialogue systems, by treating dialogue

as the stepwise construction of an object of a given type. The work in Paper III uses this idea to build complete spoken language dialogue systems.

Ljunglöf and Larsson (2008) build on the work by Ranta and Cooper (2004), and describe how GF abstract syntax can be used to specify GoDiS (Larsson 2002) Information State Update dialogue systems. This makes the dialogue management more sophisticated than that described in Paper III, but the resulting systems cannot be run in standard VoiceXML interpreters.

11.2 Compiler-like Grammar Development

The Regulus grammar compiler (Rayner et al. 2006b) generates speech recognition grammars, with the possibility of embedding basic semantic interpretation, from unification grammars. Regulus has been used in several interactive speech applications, including the MedSLT (Bouillon et al. 2005; Rayner et al. 2006a) speech translator, and the Clarissa (Rayner et al. 2005b,c) dialogue system.

UNIANCE (Bos 2002) is another system for compiling unification grammars to speech recognition grammars. It includes interpretation code for compositional semantics in the generated grammars.

The SGStudio (Wang and Acero 2005) grammar authoring tool uses a hybrid model for development of speech recognition language models and semantic interpretation. A library of parametrized grammars are used for slot-filling, while a statistical model handles the non-slot-filling parts of user input.

ARIADNE (Denecke 2002) is a dialogue system architecture for rapid prototyping. To build a dialogue system, the developer creates an ontology, parsing grammars, generation templates, database conversion rules, and a description of the services offered by the system. In ARIADNE, the developer writes a set of declarative specifications which together are used in a fixed dialogue system architecture, whereas we generate a number of components, which can be used separately, from a specification in a single formalism. Compared to ARIADNE, we lack the generic dialogue management and database interface components. On the other hand, we have support for multimodality, and GF is more linguistically expressive than the context-free grammars and output templates that ARIADNE uses.

11.3 Embedded Languages

One alternative to implementing a complete special-purpose language such as GF is to create a *domain-specific embedded language (DSEL)* (Hudak 1996). One view of a DSEL is as a complete special-purpose language that takes advantage of the existing infrastructure in an expressive general purpose language. Another view is that a DSEL is simply a software library with an interface that makes it appear to be a special-purpose language. The embedded language approach allows rapid development of the language itself, makes it possible to take advantage of the existing development tools for the host language, and allows for very good integration with programs written in the host language. On the other hand, it can be difficult to create alternative implementations of

embedded languages, to compile them to other formats, and to achieve good performance. GF was initially an embedded language (first in ALF, see Ranta 1995, then in SML, then in Haskell), before it was implemented as a stand-alone language. Most of the work in this thesis depends on it being possible to inspect GF grammars and compile them to other formats, something which would be much harder if GF were still an embedded language.

Prolog is a popular choice for creating embedded languages for natural language applications. The Core Language Engine (CLE) (Alshawi 1992) is perhaps the most ambitious such project. GF's Resource Grammar Library has a coverage comparable to that of CLE, but for a larger number of languages. In contrast to GF, CLE does not have a shared abstract syntax level. Applications can use for CLE for a number of tasks, including parsing and generation. One notable feature of the CLE that is still missing from GF is the support for packed ambiguity representations.

11.4 Interactive Development Environments for Dialogue Systems

There has been substantial work on graphical tools for semi-automatic construction for dialogue systems. One example is the CSLU Rapid Application Developer (McTear 1999), which has support for multilinguality (Cole et al. 1999). The Application Generation Platform (AGP) (Hamerich et al. 2004) can generate multilingual and multimodal interfaces to existing databases semi-automatically. DUDE (Lemon and Liu 2006) is an environment for dialogue system development where the user can semi-automatically construct a dialogue system based on a Business Process Model. DUDE generates GF grammars and makes use of the work described in this thesis to generate speech recognition grammars for the Nuance and HTK speech recognizers. DiaMant (Fliedner and Bobbert 2003) is a GUI tool for rapid development of dialogue systems based on finite state dialogue models extended with variables. Variant Transduction (Alshawi and Douglas 2001) is an example-based approach to rapid spoken language interface development. Interaction Builder (Katsurada et al. 2005) is a GUI tool for constructing web-based multimodal applications.

These tools all use graphical user interfaces to construct complete dialogue systems, whereas we use a grammar formalism as the user interface, and create general components which can also be used in other kinds of interactive speech applications, such as speech translators.

12 Future work

The overall goal is to make it easier to develop grammar-based natural language applications by generating as much as possible from GF grammars. These are some areas that could be explored further:

- A programming language for abstract syntax tree transformations, which could be used to implement application specific functionality.

- Robust parsing algorithms for GF grammars. This would allow us to take advantage of the possibility of generating statistical language models from GF grammars (Jonson 2006).
- Development of large-scale demonstration systems using our methods.

References

Hiyan Alshawi. *The Core Language Engine*. ACL-MIT Series in Natural Language Processing. MIT Press, Cambridge, Mass., May 1992. ISBN 0262011263.

Hiyan Alshawi and Shona Douglas. Variant transduction: a method for rapid development of interactive spoken interfaces. In *Proceedings of the Second SIGdial Workshop on Discourse and Dialogue*, pages 1–9, Morristown, NJ, USA, 2001. Association for Computational Linguistics. doi: 10.3115/1118078.1118080.

Johan Bos. Compilation of unification grammars with compositional semantics to speech recognition packages. In *Proceedings of the 19th international conference on Computational linguistics (COLING 2002)*, pages 1–7, Morristown, NJ, USA, 2002. Association for Computational Linguistics. doi: 10.3115/1072228.1072323.

P. Bouillon, M. Rayner, N. Chatzichrisafis, B. A. Hockey, M. Santaholma, M. Starlander, H. Isahara, K. Kanzaki, and Y. Nakao. A generic Multi-Lingual Open Source Platform for Limited-Domain Medical Speech Translation. In *Proceedings of the tenth Conference on European Association of Machine Translation (EAMT 2005), Budapest, Hungary*, pages 5–58, May 2005. URL `http://www.issco.unige.ch/pub/MedSLT_demo_EAMT05_final.pdf`.

Daniel Branca, Patsy Trench, and Dave Angus. Planet-planering. *Kalle Anka & C:o*, 1987(31), July 1987. ISSN 0345-6048. Disney story code D 8560.

Ronald A. Cole, Ben Serridge, John-Paul Hosom, Andrew Cronk, and Ed Kaiser. A Platform for Multilingual Research in Spoken Dialogue Systems. In *Proceedings of the Workshop on Multi-Lingual Interoperability in Speech Technology (MIST)*, pages 43–48, Leusden, The Netherlands, September 1999. URL `http://www.cslu.ogi.edu/people/hosom/pubs/cole_MIST-platform_1999.pdf`.

Haskell B. Curry. Some Logical Aspects of Grammatical Structure. In Roman O. Jakobson, editor, *Structure of Language and its Mathematical Aspects, volume 12 of Symposia on Applied Mathematics*, pages 56–68. American Mathematical Society, Providence, 1961.

Matthias Denecke. Rapid prototyping for spoken dialogue systems. In *Proceedings of the 19th international conference on Computational linguistics (COLING 2002)*, pages 1–7, Morristown, NJ, USA, 2002. Association for Computational Linguistics. doi: 10.3115/1072228.1072375.

Stina Ericsson, Gabriel Amores, Björn Bringert, Håkan Burden, Ann-Charlotte Forslund, David Hjelm, Rebecca Jonson, Staffan Larsson, Peter Ljunglöf, Pilar Manchón, David Milward, Guillermo Pérez, and Mikael Sandin. Software illustrating a unified approach to multimodality and multilinguality in the in-home domain. deliverable 1.6, 2006. URL `http://www.talk-project.org/fileadmin/talk/publications_public/deliverables_public/D1_6.pdf`.

Gerhard Fliedner and Daniel Bobbert. DiaMant: A Tool for Rapidly Developing Spoken Dialogue Systems. In *Proceedings of the 7th Workshop on the Semantics and Pragmatics of Dialogue (DiaBruck), Wallerfangen, Germany*, 2003. URL `http://www.coli.uni-saarland.de/conf/diabruck/submission_finals/abstracts/320/demo_320.pdf`.

Stefan Hamerich, Volker Schubert, Volker Schless, Ricardo de Córdoba, José M. Pardo, Luis F. d'Haro, Basilis Kladis, Otilia Kocsis, and Stefan Igel. Semi-Automatic Generation of Dialogue Applications in the GEMINI Project. In Michael Strube and Candy Sidner, editors, *Proceedings of the 5th SIGdial Workshop on Discourse and Dialogue*, pages 31–34, Cambridge, Massachusetts, USA, 2004. Association for Computational Linguistics. URL `http://acl.ldc.upenn.edu/hlt-naacl2004/sigdial04/pdf/hamerich.pdf`.

Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es), 1996. doi: 10.1145/242224.242477.

Rebecca Jonson. Generating Statistical Language Models from Interpretation Grammars in Dialogue Systems. In *EACL 2006, 11st Conference of the European Chapter of the Association for Computational Linguistics*, 2006. URL `http://acl.ldc.upenn.edu/E/E06/E06-1008.pdf`.

Rebecca Jonson. Grammar-based context-specific statistical language modelling. In *Proceedings of the Workshop on Grammar-Based Approaches to Spoken Language Processing*, pages 25–32, Prague, Czech Republic, June 2007. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W07/W07-1804`.

K. Katsurada, H. Adachi, K. Sato, H. Yamada, and T. Nitta. Interaction builder: A rapid prototyping tool for developing web-based MMI applications. *IEICE Trans Inf Syst*, E88-D(11):2461–2467, 2005. doi: 10.1093/ietisy/e88-d.11.2461.

Staffan Larsson. *Issue-based Dialogue Management*. PhD thesis, Göteborg University, Göteborg, Sweden, 2002.

Oliver Lemon and Xingkun Liu. DUDE: a Dialogue and Understanding Development Environment, mapping Business Process Models to Information State Update dialogue systems. In *EACL 2006, 11st Conference of the European Chapter of the Association for Computational Linguistics*, 2006. URL `http://www.aclweb.org/anthology-new/E/E06/E06-2004.pdf`.

Peter Ljunglöf and Staffan Larsson. A Grammar Formalism for Specifying ISU-Based Dialogue Systems. In *Advances in Natural Language Processing, 6th International Conference, GoTAL 2008, Gothenburg, Sweden*, volume 5221 of *Lecture Notes in Computer Science*, pages 303–314. Springer, August 2008. doi: 10.1007/978-3-540-85287-2_29.

Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984.

Michael F. McTear. Software to support research and development of spoken dialogue systems. In *Proceedings, Sixth European Conference on Speech Communication and Technology (EUROSPEECH'99), Budapest, Hungary*, pages 339–342. ISCA Archive, September 1999. URL `http://www.cslu.ogi.edu/toolkit/pubs/pdf/mctear_EUROSPEECH_99.pdf`.

Moisés S. Meza Moreno. Implementation of a JavaScript Syntax Editor and Parser for Grammatical Framework. Master's thesis, Chalmers University of Technology, 2008.

Nadine Perera and Aarne Ranta. Dialogue System Localization with the GF Resource Grammar Library. In *Proceedings of the Workshop on Grammar-Based Approaches to Spoken Language Processing, Prague, Czech Republic*, pages 17–24. Association for Computational Linguistics, 2007. URL `http://www.aclweb.org/anthology/W/W07/W07-1803`.

Roberto Pieraccini and Juan Huerta. Where do we go from here? Research and commercial spoken dialog systems. In *Proceedings of the 6th SIGdial Workshop on Discourse and Dialogue*, Lisbon, Portugal, September 2005. URL `http://www.sigdial.org/workshops/workshop6/proceedings/pdf/65-SigDial2005_8.pdf`.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004. ISSN 0956-7968. doi: 10.1017/S0956796803004738.

Aarne Ranta. Modular Grammar Engineering in GF. *Research on Language and Computation*, 5(2):133–158, June 2007. doi: 10.1007/s11168-007-9030-6.

Aarne Ranta. Grammars as software libraries. In Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin, editors, *From semantics to computer science: essays in honor of Gilles Kahn*. Cambridge University Press, 2008. URL `http://www.cs.chalmers.se/~aarne/articles/libraries-kahn.pdf`.

Aarne Ranta. Syntactic categories in the language of mathematics. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs, International Workshop TYPES '94, Båstad, Sweden, June 6-10, 1994*, volume 996 of *Lecture Notes in Computer Science*, pages 162–182, Heidelberg, 1995. Springer. doi: 10.1007/3-540-60579-7_9.

Aarne Ranta and Robin Cooper. Dialogue Systems as Proof Editors. *Journal of Logic, Language and Information*, 13(2):225–240, 2004. ISSN 0925-8531. doi: 10.1023/B:JLLI.0000024736.34644.48.

Manny Rayner, David Carter, Pierrette Bouillon, Vassilis Digalakis, and Mats Wirén, editors. *The Spoken Language Translator*. Studies in Natural Language Processing. Cambridge University Press, Cambridge, UK, November 2000. doi: 10.2277/0521770777.

Manny Rayner, Pierrette Bouillon, Nikos Chatzichrisafis, Beth A. Hockey, Marianne Santaholma, Marianne Starlander, Hitoshi Isahara, Kyoko Kanzaki, and Yukie Nakao. A Methodology for Comparing Grammar-Based and Robust Approaches to Speech Understanding. In *Proceedings of Interspeech 2005*, 2005a. URL `http://www.issco.unige.ch/pub/RaynerEAInterspeech2005.pdf`.

Manny Rayner, Beth A. Hockey, Nikos Chatzichrisafis, Kim Farrell, and Jean-Michel Renders. A voice enabled procedure browser for the International Space Station. In *Proceedings of the ACL Interactive Poster and Demonstration Sessions (ACL 2005), Ann Arbor, Michigan*, pages 29–32, Morristown, NJ, USA, June 2005b. Association for Computational Linguistics. doi: 10.3115/1225753.1225761.

Manny Rayner, Beth A. Hockey, Jean-Michel Renders, Nikos Chatzichrisafis, and Kim Farrell. Spoken Language Processing in the Clarissa Procedure Browser. Technical report, International Computer Science Institute, Berkeley, California, April 2005c. URL `ftp://ftp.icsi.berkeley.edu/pub/techreports/2005/tr-05-005.pdf`.

Manny Rayner, Pierrette Bouillon, Nikos Chatzichrisafis, Marianne Santaholma, Marianne Starlander, Beth A. Hockey, Yukie Nakao, Hitoshi Isahara, and Kyoko Kanzaki. MedSLT: A Limited-Domain Unidirectional Grammar-Based Medical Speech Translator. In *Proceedings of the First International Workshop on Medical Speech Translation*, pages 40–43, New York, New York, 2006a. Association for Computational Linguistics. URL `http://acl.ldc.upenn.edu/W/W06/W06-3707.pdf`.

Manny Rayner, Beth A. Hockey, and Pierrette Bouillon. *Putting Linguistics into Speech Recognition: The Regulus Grammar Compiler*. CSLI Publications, Ventura Hall, Stanford University, Stanford, CA 94305, USA, July 2006b. ISBN 1575865262.

Anders Christian Sivebæk, François Willot, and Lars Jensen. IRC conversation, March 2007. `irc://irc.inducks.org:6667/dcml`.

Tony Strobl and Steve Steere. Operation Hajön. *Musse Pigg & C:o*, 1985(5), May 1985. ISSN 0349-1463. Disney story code S 81111.

Alex Waibel. Speech Translation: Past, Present and Future. In *INTERSPEECH-2004*, pages 353–356, October 2004.

Ye-Yi Wang and Alex Acero. SGStudio: Rapid Semantic Grammar Development for Spoken Language Understanding. In *Proceedings of the Interspeech Conference*, Lisbon, Portugal, September 2005. URL `http://research.microsoft.com/srg/papers/2005-yeyiwang-eurospeech.pdf`.

Paper I

Speech Recognition Grammar Compilation in Grammatical Framework

SPEECHGRAM 2007, Prague

Speech Recognition Grammar Compilation in Grammatical Framework

Björn Bringert
Department of Computer Science and Engineering
Chalmers University of Technology
and University of Gothenburg
SE-412 96 Göteborg, Sweden
bringert@chalmers.se

Abstract

This paper describes how grammar-based language models for speech recognition systems can be generated from Grammatical Framework (GF) grammars. Context-free grammars and finite-state models can be generated in several formats: GSL, SRGS, JSGF, and HTK SLF. In addition, semantic interpretation code can be embedded in the generated context-free grammars. This enables rapid development of portable, multilingual and easily modifiable speech recognition applications.

1 Introduction

Speech recognition grammars are used for guiding speech recognizers in many applications. However, there are a number of problems associated with writing grammars in the low-level, system-specific formats required by speech recognizers. This work addresses these problems by generating speech recognition grammars and semantic interpretation components from grammars written in Grammatical Framework (GF), a high-level, type-theoretical grammar formalism. Compared to existing work on compiling unification grammars, such as Regulus (Rayner et al. 2006), our work uses a type-theoretical grammar formalism with a focus on multilinguality and modular grammar development, and supports multiple speech recognition grammar formalisms, including finite-state models.

We first outline some existing problems in the development and maintenance of speech recognition grammars, and describe how our work attempts to address these problems. In the following two sections we introduce speech recognition grammars and Grammatical Framework. The bulk of the paper then describes how we generate context-free speech recognition grammars, finite-state language models and semantic interpretation code from GF grammars. We conclude by

giving references to a number of experimental dialogue systems which already use our grammar compiler for generating speech recognition grammars.

Expressivity Speech recognition grammars are written in simple formalisms which do not have the powerful constructs of high-level grammar formalisms. This makes speech recognition grammar writing labor-intensive and error prone, especially for languages with more inflection and agreement than English.

This is solved by using a high-level grammar formalism with powerful constructs and a grammar library which implements the domain-independent linguistic details.

Duplicated work When speech recognition grammars are written directly in the low-level format required by the speech recognizer, other parts of the system, such as semantic interpretation components, must often be constructed separately.

This duplicated work can be avoided by generating all the components from a single declarative source, such as a GF grammar.

Consistency Because of the lack of abstraction mechanisms and consistency checks, it is difficult to modify a system which uses hand-written speech recognition grammars. The problem is multiplied when the system is multilingual. The developer has to modify the speech recognition grammar and the semantic interpretation component manually for each language. A simple change may require touching many parts of the grammar, and there are no automatic consistency checks.

The strong typing of the GF language enforces consistency between the semantics and the concrete representation in each language.

Localization With hand-written grammars, it is about as difficult to add support for a new language as it is to write the grammar and semantic interpretation for the first language.

GF's support for multilingual grammars and the common interface implemented by all grammars in the GF resource grammar library makes it easier to translate a grammar to a new language.

Portability A grammar in any given speech recognition grammar format cannot be used with a speech recognizer which uses another format.

In our approach, a GF grammar is used as the canonical representation which the developer works with, and speech recognition grammars in many formats can be generated automatically from this representation.

2 Speech Recognition Grammars

To achieve acceptable accuracy, speech recognition software is guided by a *language model* which defines the language which can be recognized. A language

model may also assign different probabilities to different strings in the language. A language model can either be a *statistical language model (SLM)*, such as an n-gram model, or a *grammar-based language model*, for example a *context-free grammar (CFG)* or a *finite-state automaton (FSA)*. In this paper, we use the term *speech recognition grammar (SRG)* to refer to all grammar-based language models, including context-free grammars, regular grammars and finite-state automata.

3 Grammatical Framework

Grammatical Framework (GF) (Ranta 2004) is a grammar formalism based on constructive type theory. In GF, an *abstract syntax* defines a semantic representation. A *concrete syntax* declares how terms in an abstract syntax are *linearized*, that is, how they are mapped to concrete representations. GF grammars can be made multilingual by having multiple concrete syntaxes for a single abstract syntax.

3.1 The Resource Grammar Library

The GF Resource Grammar Library (Ranta et al. 2006) currently implements the morphological and syntactic details of 10 languages. This library is intended to make it possible to write grammars without caring about the linguistic details of particular languages. It is inspired by *library-based software engineering*, where complex functionality is implemented in reusable software libraries with simple interfaces.

The resource grammar library is used through GF's facility for *grammar composition*, where the abstract syntax of one grammar is used in the implementation of the concrete syntax of another grammar. Thus, an application grammar writer who uses a resource grammar uses its abstract syntax terms to implement the linearizations in the application grammar.

The resource grammars for the different languages implement a common interface, i.e. they all have a common abstract syntax. This means that grammars which are implemented using resource grammars can be easily localized to other languages. Localization normally consists of translating the application-specific lexical items, and adjusting any linearizations which turn out to be unidiomatic in the language in question. For example, when the GoTGoDiS (Ericsson et al. 2006) application was localized to Finnish, only 3 out of 180 linearization rules had to be changed.

3.2 An Example GF Grammar

Figure 1 contains a small example GF abstract syntax. Figure 2 defines an English concrete syntax for it, using the resource grammar library. We will use this grammar when we show examples of speech recognition grammar generation later.

```
abstract Food = {
  flags startcat = Order;
  cat Order;
      Items;
      Item;
      Number;
      Size;
  fun order : Items → Order;
      and   : Items → Items → Items;
      items : Item → Number → Size → Items;
      pizza, beer   : Item;
      one, two      : Number;
      small, large : Size;
}
```

Figure 1. `Food.gf`: A GF abstract syntax module.

In the abstract syntax, **cat** judgements introduce syntactic categories, and **fun** judgements declare constructors in those categories. For example, the *items* constructor makes an Items term from an Item, a Number and a Size. The term *items pizza two small* is an example of a term in this abstract syntax.

In the concrete syntax, a **lincat** judgement declares the type of the concrete terms generated from the abstract syntax terms in a given category. The linearization of each constructor is declared with a **lin** judgement. In the concrete syntax in Figure 2, library functions from the English resource grammar are used for the linearizations, but it is also possible to write concrete syntax terms directly. The linearization of the term *items pizza two small* is $\{s =$ "two small pizzas"$\}$, a record containing a single string field.

By changing the imports and the four lexical items, this grammar can be translated to any other language for which there is a resource grammar. For example, in the German version, we replace (*regN* "beer") with (*reg2N* "Bier" "Biere" *neuter*) and so on. The functions *regN* and *reg2N* implement paradigms for regular English and German nouns, respectively. This replacement can be formalized using GF's *parametrized modules*, which lets one write a common implementation that can be instantiated with the language-specific parts. Note that the application grammar does not deal with details such as agreement, as this is taken care of by the resource grammar.

```
concrete FoodEng of Food = open TryEng in {
  lincat Order  = Utt;
         Items  = NP;
         Item   = CN;
         Number = Det;
         Size   = AP;
  lin order x    = mkUtt x;
      and x y    = mkNP and_Conj x y;
      items x n s = mkNP n (mkCN s x);
      pizza      = mkCN (regN "pizza");
      beer       = mkCN (regN "beer");
      one        = mkDet n1_Numeral;
      two        = mkDet n2_Numeral;
      small      = mkAP (regA "small");
      large      = mkAP (regA "large");
}
```

Figure 2. `FoodEng.gf`: English concrete syntax for the abstract syntax in Figure 1.

4 Generating Context-free Grammars

4.1 Algorithm

GF grammars are converted to context-free speech recognition grammars in a number of steps. An overview of the compilation pipeline is show in Figure 3. The figure also includes compilation to finite-state automata, as described in Section 5. Each step of the compilation is described in more detail in the sections below.

PGF compilation The linearization rules in the GF grammar are first compiled to PGF (Portable Grammar Format, Angelov et al. 2008) linearization rules.

PMCFG compilation The PGF linearization rules are then compiled to a Parallel Multiple Context-Free Grammar (PMCFG) (Angelov et al. 2008).

CFG conversion The PMCFG is approximated with a CFG by converting each PMCFG category-field pair to a CFG category.

Cycle elimination All directly and indirectly cyclic productions are removed, since they cannot be handled gracefully by the subsequent left-recursion elimination. Such productions do not contribute to the coverage to the grammar, only to the set of possible semantic results.

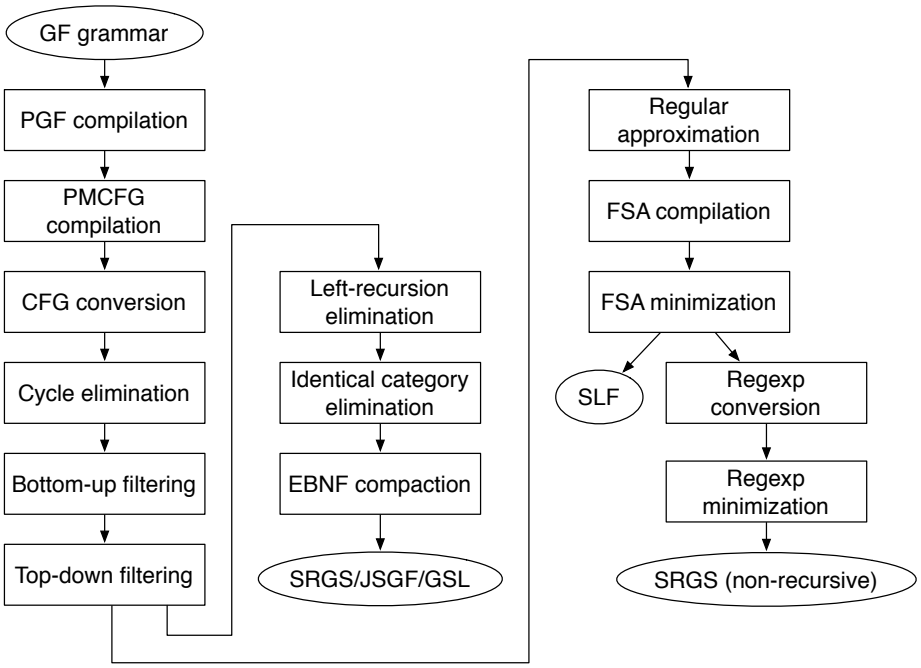

Figure 3. Grammar compilation pipeline.

Bottom-up filtering Productions whose right-hand sides use categories for which there are no productions are removed, since these will never match any input.

Top-down filtering Only productions for categories which can be reached from the start category are kept. This is mainly used to remove parts of the grammar which are unused because of the choice of start category. One example where this is useful is when a speech recognition grammar is generated from a multimodal grammar (Bringert et al. 2005). In this case, the start category is different from the start category used by the parser, in that its linearization only contains the speech component of the input. Top-down filtering then has the effect of excluding the non-speech modalities from the speech recognition grammar.

Left-recursion elimination All direct and indirect left-recursion is removed using the LC_{LR} transform described by Moore (2000). We have modified the LC_{LR} transform to avoid adding productions which use a category A–X when there are no productions for A–X.

Identical category elimination In this step, the categories are grouped into equivalence classes by their right-hand sides and semantic annotations. The

categories $A_1 \ldots A_n$ in each class are replaced by a single category $A_1 + \ldots + A_n$ throughout the grammar, discarding any duplicate productions. This has the effect of replacing all categories which have identical sets of productions with a single category. Concrete syntax parameters which do not affect inflection is one source of such redundancy; the LC_{LR} transform is another.

EBNF compaction The resulting context-free grammar is compacted into an *Extended Backus-Naur Form (EBNF)* representation. This reduces the size and improves the readability of the final grammar. The compaction is done by, for each category, grouping all the productions which have the same semantic interpretation, and the same sequence of non-terminals on their right-hand sides, ignoring any terminals. The productions in each group are merged into one EBNF production, where the terminal sequences between the non-terminals are converted to regular expressions which are the unions of the original terminal sequences. These regular expressions are then minimized.

Conversion to output format The resulting non-left-recursive grammar is converted to SRGS, JSGF or Nuance GSL format.

A fragment of a SRGS ABNF grammar generated from the GF grammar in Figure 2 is shown below. The left-recursive *and* rule was removed from the grammar before compilation, as the left-recursion elimination step makes it difficult to read the generated grammar. The fragment shown here is for the singular part of the *items* rule.

```
$FE1 = $FE6 $FE9 $FE4;
$FE6 = one;
$FE9 = large | small;
$FE4 = beer  | pizza;
```

The corresponding fragment generated from the German version of the grammar is more complex, since the numeral and the adjective must agree with the gender of the noun.

```
$FG1  = $FG10 $FG13 $FG6 | $FG9 $FG12 $FG4;
$FG9  = eine;
$FG10 = ein;
$FG12 = große  | kleine;
$FG13 = großes | kleines;
$FG4  = Pizza;
$FG6  = Bier;
```

4.2 Discussion

The generated grammar is an overgenerating approximation of the original GF grammar. This is inevitable, since the GF formalism is stronger than context-free grammars, for example through its support for reduplication. GF's support for dependently typed and higher-order abstract syntax is also not yet carried over to the generated speech recognition grammars. This could be handled

in a subsequent semantic interpretation step. However, that requires that the speech recognizer considers multiple hypotheses, since some may be discarded by the semantic interpretation. Currently, if the abstract syntax types are only dependent on finite types, the grammar can be expanded to remove the dependencies. This appears to be sufficient for many realistic applications.

In some cases, empty productions in the generated grammar could cause problems for the cycle and left-recursion elimination, though we have yet to encounter this in practice. Empty productions can be removed by transforming the grammar, though this has not yet been implemented.

For some grammars, the initial CFG generation can generate a very large number of productions. While the resulting speech recognition grammars are of a reasonable size, the large intermediate grammars can cause memory problems. Further optimization is needed to address this problem.

5 Finite-State Models

5.1 Algorithm

Some speech recognition systems use finite-state automata rather than context-free grammars as language models. GF grammars can be compiled to finite-state automata using the procedure shown in Figure 3. The initial part of the compilation to a finite-state model is shared with the context-free SRG compilation, and is described in Section 4.

Regular approximation The context-free grammar is approximated with a regular grammar, using the algorithm described by Mohri and Nederhof (2001).

Compilation to finite-state automata The regular grammar is converted into a set of *non-deterministic finite automata (NFA)* using a modified version of the *make_fa* algorithm described by Nederhof (2000). For realistic grammars, applying the original *make_fa* algorithm to the whole grammar generates a very large automaton, since a copy of the sub-automaton corresponding to a given category is made for every use of the category.

Instead, one automaton is generated for each category in the regular grammar. All categories which are not in the same mutually recursive set as the category for which the automaton is generated are treated as terminal symbols. This results in a set of automata with edges labeled with either terminal symbols or the names of other automata.

If desired, the set of automata can be converted into a single automaton by substituting each category-labeled edge with a copy of the corresponding automaton. Note that this always terminates, since the sub-automata do not have edges labeled with the categories from the same mutually recursive set.

Minimization Each of the automata is turned into a minimal *deterministic finite automaton (DFA)* by using Brzozowski's (1962) algorithm, which minimizes the automaton by performing two determinizations and reversals.

Conversion to output format The resulting finite automaton can be output in HTK *Standard Lattice Format (SLF)*. SLF supports sub-lattices, which allows us to convert our set of automata directly into a set of lattices. Since SLF uses labeled nodes, rather than labeled edges, we move the labels to the nodes. This is done by first introducing a new labeled node for each edge, and then eliminating all internal unlabeled nodes. Figure 4 shows the SLF model generated from the example grammar. For clarity, the sub-lattices have been inlined.

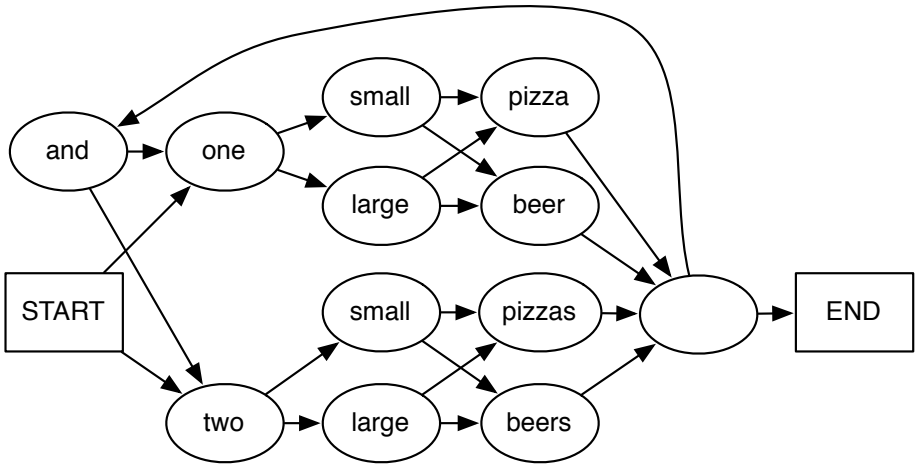

Figure 4. SLF model generated from the grammar in Figure 2.

Regular expression conversion and minimization Some speech recognizers, such as Nuance Recognizer 9.0, only support non-recursive SRGS grammars. We build such grammars by starting from the DFAs produced for each mutually recursive set in the previous step. Each DFA is converted to a regular expression which is then minimized. The end result is an EBNF grammar without any recursion, which can be printed in SRGS format.

5.2 Discussion

Finite-state models are even more restrictive than context-free grammars. This problem is handled by approximating the context-free grammar with an overgenerating finite-state automaton. This may lead to failure in a subsequent parsing step, which, as in the context-free case, is acceptable if the recognizer can return all hypotheses.

6 Semantic Interpretation

Semantic interpretation can be done as a separate parsing step after speech recognition, or it can be done with semantic information embedded in the speech recognition grammar. The latter approach resembles the *semantic actions* used by parser generators for programming languages. One formalism for semantic interpretation is the proposed *Semantic Interpretation for Speech Recognition (SISR)* standard. SISR tags are pieces of JavaScript code embedded in the speech recognition grammar.

6.1 Algorithm

The GF system can include SISR tags when generating speech recognitions grammars in SRGS and JSGF format. The SISR tags are generated from the tree building information in the PMCFG. The result of the semantic interpretation is an abstract syntax term.

The left-recursion elimination step makes it somewhat challenging to produce correct abstract syntax trees. We have extended Moore's (2000) LC_{LR} transform to preserve the semantic interpretation. The LC_{LR} transform introduces new categories of the form $A{-}X$ where X is a proper left corner of a category A. The new category $A{-}X$ can be understood as "the category A, but missing an initial X". Thus the semantic interpretation for a production in $A{-}X$ is the semantic interpretation for the original A-production, abstracted (in the λ-calculus sense) over the semantic interpretation of the missing X. Conversely, where-ever a category $A{-}X$ is used, its result is applied to the interpretation of the occurrence of X.

6.2 Discussion

As discussed in Section 4.2, the semantic interpretation code could be used to implement the non-context-free features of GF, but this is not yet done.

The slot-filling mechanism in the GSL format could also be used to build semantic representations, by returning program code which can then be executed. The UNIANCE grammar compiler (Bos 2002) uses that approach.

7 Related Work

7.1 Unification Grammar Compilation

Compilation of unification grammars to speech recognition grammars is well described in the literature (Moore 1999; Dowding et al. 2001). Regulus (Rayner et al. 2006) is perhaps the most ambitious such system. Like GF, Regulus uses a general grammar for each language, which is specialized to a domain-specific one. Ljunglöf (Ljunglöf 2007) relates GF and Regulus by showing how to convert GF grammars to Regulus grammars. We carry compositional semantic interpretation through left-recursion elimination using the same idea as

the UNIANCE grammar compiler (Bos 2002), though our version handles both direct and indirect left-recursion.

The main difference between our work and the existing compilers is that we work with type-theoretical grammars rather than unification grammars. While the existing work focuses on GSL as the output language, we also support a number of other formats, including finite-state models. By using the GF resource grammars, speech recognition language models can be produced for more languages than with previous systems. One shortcoming of our system is that it does not yet have support for weighted grammars.

7.2 Generating SLMs from GF Grammars

Jonson (2006) has shown that in addition to generating grammar-based language models, GF can be used to build statistical language models (SLMs). It was found that compared to our grammar-based approach, use of generated SLMs improved the recognition performance for out-of-grammar utterances significantly.

8 Results

Speech recognition grammars generated from GF grammars have already been used in a number of research dialogue systems.

GOTTIS (Bringert et al. 2005; Ericsson et al. 2006), an experimental multimodal and multilingual dialogue system for public transportation queries, uses GF grammars for parsing multimodal input. For speech recognition, it uses GSL grammars generated from the speech modality part of the GF grammars.

DJ-GoDiS, GoDiS-deLUX, and GoTGoDiS (Ericsson et al. 2006) are three applications which use GF grammars for speech recognition and parsing together with the GoDiS implementation of issue-based dialogue management (Larsson 2002). GoTGoDiS has been translated to 7 languages using the GF resource grammar library, with each new translation taking less than one day (Ericsson et al. 2006).

The DICO (Villing and Larsson 2006) dialogue system for trucks has recently been modified to use GF grammars for speech recognition and parsing (Ljunglöf 2007).

DUDE (Lemon and Liu 2006) and its extension REALL-DUDE (Lemon et al. 2006) are environments where non-experts can develop dialogue systems based on Business Process Models describing the applications. From keywords, prompts and answer sets defined by the developer, the system generates a GF grammar. This grammar is used for parsing input, and for generating a language model in SLF or GSL format.

The Voice Programming system by Georgila and Lemon (2006) uses an SLF language model generated from a GF grammar.

Perera and Ranta (2007) have studied how GF grammars can be used for localization of dialogue systems. A GF grammar was developed and localized

to 4 other languages in significantly less time than an equivalent GSL grammar. They also found the GSL grammar generated by GF to be much smaller than the hand-written GSL grammar.

9 Conclusions

We have shown how GF grammars can be compiled to several common speech recognition grammar formats. This has helped decrease development time, improve modifiability, aid localization and enable portability in a number of experimental dialogue systems.

Several systems developed in the TALK and DICO projects use the same GF grammars for speech recognition, parsing and multimodal fusion (Ericsson et al. 2006). Using the same grammar for multiple system components **reduces development and modification costs**, and makes it easier to **maintain consistency** within the system.

The feasibility of **rapid localization** of dialogue systems which use GF grammars has been demonstrated in the GoTGoDiS (Ericsson et al. 2006) system, and in experiments by Perera and Ranta (2007).

Using speech recognition grammars generated by GF makes it easy to **support different speech recognizers**. For example, by using the GF grammar compiler, the DUDE (Lemon and Liu 2006) system can support both the ATK and Nuance recognizers.

Implementations of the methods described in this paper are freely available as part of the GF distribution[1].

Acknowledgments

Aarne Ranta, Peter Ljunglöf, Rebecca Jonson, David Hjelm, Ann-Charlotte Forslund, Håkan Burden, Xingkun Liu, Oliver Lemon, and the anonymous referees have contributed valuable comments on the grammar compiler implementation and/or this article. We would like to thank Nuance Communications, Inc., OptimSys, s.r.o., and Opera Software ASA for software licenses and technical support. The code in this paper has been typeset using `lhs2TeX`, with help from Andres Löh. This work has been partly funded by the EU TALK project, IST-507802.

References

Krasimir Angelov, Björn Bringert, and Aarne Ranta. PGF: A Portable Run-Time Format for Type-Theoretical Grammars. *Journal of Logic, Language and Information*, submitted, 2008.

[1] `http://www.cs.chalmers.se/~aarne/GF/`

Johan Bos. Compilation of unification grammars with compositional semantics to speech recognition packages. In *Proceedings of the 19th international conference on Computational linguistics (COLING 2002)*, pages 1–7, Morristown, NJ, USA, 2002. Association for Computational Linguistics. doi: 10.3115/1072228.1072323.

Björn Bringert, Robin Cooper, Peter Ljunglöf, and Aarne Ranta. Multimodal Dialogue System Grammars. In *Proceedings of DIALOR'05, Ninth Workshop on the Semantics and Pragmatics of Dialogue, Nancy, France*, pages 53–60, June 2005. URL `http://dialor05.loria.fr/Papers/07-BjornBringert.pdf`.

Janusz A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical theory of Automata*, Volume 12 of MRI Symposia Series, pages 529–561. Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y., 1962.

John Dowding, Beth A. Hockey, Jean M. Gawron, and Christopher Culy. Practical issues in compiling typed unification grammars for speech recognition. In *ACL '01: Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pages 164–171, Morristown, NJ, USA, 2001. Association for Computational Linguistics. doi: 10.3115/1073012.1073034.

Stina Ericsson, Gabriel Amores, Björn Bringert, Håkan Burden, Ann-Charlotte Forslund, David Hjelm, Rebecca Jonson, Staffan Larsson, Peter Ljunglöf, Pilar Manchón, David Milward, Guillermo Pérez, and Mikael Sandin. Software illustrating a unified approach to multimodality and multilinguality in the in-home domain. deliverable 1.6, 2006. URL `http://www.talk-project.org/fileadmin/talk/publications_public/deliverables_public/D1_6.pdf`.

Kallirroi Georgila and Oliver Lemon. Programming by Voice: enhancing adaptivity and robustness of spoken dialogue systems. In *BRANDIAL'06, Proceedings of the 10th Workshop on the Semantics and Pragmatics of Dialogue*, pages 199–200, 2006. URL `http://www.ling.uni-potsdam.de/brandial/Proceedings/brandial06_georgila_etal.pdf`.

Rebecca Jonson. Generating Statistical Language Models from Interpretation Grammars in Dialogue Systems. In *EACL 2006, 11st Conference of the European Chapter of the Association for Computational Linguistics*, 2006. URL `http://acl.ldc.upenn.edu/E/E06/E06-1008.pdf`.

Staffan Larsson. *Issue-based Dialogue Management*. PhD thesis, Göteborg University, Göteborg, Sweden, 2002.

Oliver Lemon and Xingkun Liu. DUDE: a Dialogue and Understanding Development Environment, mapping Business Process Models to Information State Update dialogue systems. In *EACL 2006, 11st Conference of the European Chapter of the Association for Computational Linguistics*, 2006. URL `http://www.aclweb.org/anthology-new/E/E06/E06-2004.pdf`.

Oliver Lemon, Xingkun Liu, Daniel Shapiro, and Carl Tollander. Hierarchical Reinforcement Learning of Dialogue Policies in a development environment for dialogue systems: REALL-DUDE. In *BRANDIAL'06, Proceedings of the 10th Workshop on the Semantics and Pragmatics of Dialogue*, pages 185–186, September 2006. URL `http://www.ling.uni-potsdam.de/brandial/Proceedings/brandial06_lemon_etal.pdf`.

Peter Ljunglöf. Converting Grammatical Framework to Regulus. In *Proceedings of the Workshop on Grammar-Based Approaches to Spoken Language Processing*, pages 9–16, Prague, Czech Republic, 2007. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W07/W07-1802`.

Peter Ljunglöf. Personal communication, March 2007.

Mehryar Mohri and Mark J. Nederhof. Regular Approximation of Context-Free Grammars through Transformation. In Jean C. Junqua and Gertjan van Noord, editors, *Robustness in Language and Speech Technology*, volume 17 of *Text, Speech and Language Technology*, pages 153–163. Kluwer Academic Publishers, Dordrecht, 2001. URL `http://www.coli.uni-sb.de/publikationen/softcopies/Mohri:2001:RAC.pdf`.

Robert C. Moore. Removing left recursion from context-free grammars. In *Proceedings of the first conference on North American chapter of the Association for Computational Linguistics*, pages 249–255, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. doi: 10.1145/321250.321254.

Robert C. Moore. Using Natural-Language Knowledge Sources in Speech Recognition. In K. M. Ponting, editor, *Computational Models of Speech Pattern Processing*, pages 304–327. Springer, 1999. URL `http://research.microsoft.com/users/bobmoore/nato-asi.pdf`.

Mark J. Nederhof. Regular Approximation of CFLs: A Grammatical View. In Harry Bunt and Anton Nijholt, editors, *Advances in Probabilistic and other Parsing Technologies*, volume 16 of *Text, Speech and Language Technology*, pages 221–241. Kluwer Academic Publishers, 2000. URL `http://www.dcs.st-and.ac.uk/~mjn/publications/2000d.pdf`.

Nadine Perera and Aarne Ranta. Dialogue System Localization with the GF Resource Grammar Library. In *Proceedings of the Workshop on Grammar-Based Approaches to Spoken Language Processing, Prague, Czech Republic*, pages 17–24. Association for Computational Linguistics, 2007. URL `http://www.aclweb.org/anthology/W/W07/W07-1803`.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004. ISSN 0956-7968. doi: 10.1017/S0956796803004738.

Aarne Ranta, Ali El Dada, and Janna Khegai. The GF Resource Grammar Library, June 2006. URL http://www.cs.chalmers.se/~aarne/GF/doc/resource.pdf.

Manny Rayner, Beth A. Hockey, and Pierrette Bouillon. *Putting Linguistics into Speech Recognition: The Regulus Grammar Compiler*. CSLI Publications, Ventura Hall, Stanford University, Stanford, CA 94305, USA, July 2006. ISBN 1575865262.

Jessica Villing and Staffan Larsson. Dico: A Multimodal Menu-based In-vehicle Dialogue System. In *BRANDIAL'06, Proceedings of the 10th Workshop on the Semantics and Pragmatics of Dialogue*, pages 187–188, 2006. URL http://www.ling.uni-potsdam.de/brandial/Proceedings/brandial06_villing_etal.pdf.

Paper II | Multimodal Dialogue System Grammars

DIALOR 2005, Nancy

Multimodal Dialogue System Grammars*

Björn Bringert, Peter Ljunglöf, Aarne Ranta
Department of Computer Science and Engineering
Chalmers University of Technology
and University of Gothenburg
{bringert,peb,aarne}@chalmers.se

Robin Cooper
Department of Linguistics
University of Gothenburg
cooper@ling.gu.se

Abstract

We describe how multimodal grammars for dialogue systems can be written using the Grammatical Framework (GF) formalism. A proof-of-concept dialogue system constructed using these techniques is also presented. The software engineering problem of keeping grammars for different languages, modalities and systems (such as speech recognizers and parsers) in sync is reduced by the formal relationship between the abstract and concrete syntaxes, and by generating equivalent grammars from GF grammars.

1 Introduction

We are interested in building multilingual multimodal grammar-based dialogue systems which are clearly recognisable to users as the same system even if they use the system in different languages or in different domains using different mixes of modalities (e.g. in-house vs in-car, and within the in-house domain with vs without a screen for visual interaction and touch/click input). We wish to be able to guarantee that the functionality of the system is the same under the different conditions.

Our previous experience with building such multilingual dialogue systems is that there is a software engineering problem keeping the linguistic coverage in sync for different languages. This problem is compounded by the fact that for each language it is normally the case that a dialogue system requires more than one grammar, e.g. one grammar for speech recognition and another for

* This project is supported by the EU project TALK (Talk and Look, Tools for Ambient Linguistic Knowledge), IST-507802.

interaction with the dialogue manager. Thus multilingual systems become very difficult to develop and maintain.

In this paper we will explain the nature of the Grammatical Framework (GF) and how it may provide us with a solution to this problem. The system is oriented towards the writing of multilingual and multimodal grammars and forces the grammar writer to keep a collection of grammars in sync. It does this by using computer science notions of abstract and concrete syntax. Essentially abstract syntax corresponds to the domain knowledge representation of the system and several concrete syntaxes characterising both natural language representations of the domain and representations in other modalities are related to a single abstract syntax.

GF has a type checker that forces concrete syntaxes to give complete coverage of the abstract syntax and thus will immediately tell the grammar writer if the grammars are not in sync. In addition the framework provides possibilities for converting from one grammar format to another and for combining grammars and extracting sub-grammars from larger grammars.

2 The Grammatical Framework and multilingual grammars

The main idea of Grammatical Framework (GF) is the separation of abstract and concrete syntax. The abstract part of a grammar defines a set of abstract syntactic structures, called abstract terms or trees; and the concrete part defines a relation between abstract structures and concrete structures.

As an example of a GF representation, the following abstract syntax tree represents a possible user input in our example dialogue system.

GoFromTo (PStop Chalmers) (PStop Valand)

The English concrete syntax relates the query to the string

"I want to go from Chalmers to Valand"

The Swedish concrete syntax relates it to the string

"Jag vill åka från Chalmers till Valand"

The strings are generated from the tree in a compositional rule-to-rule fashion. The generation rules are automatically inverted to parsing rules.

The abstract theory of Grammatical Framework (Ranta 2004) is a version of dependent type theory, similar to LF (Harper et al. 1993), ALF (Magnusson and Nordström 1994) and COQ (Coq). What GF adds to the logical framework is the possibility of defining concrete syntax. The expressiveness of the concrete syntax has developed into a functional programming language, similar to a restricted version of programming languages like Haskell (Peyton Jones 2003) and ML (Milner et al. 1997).

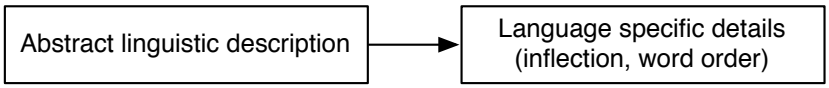

Figure 1. Higher-level language descriptions

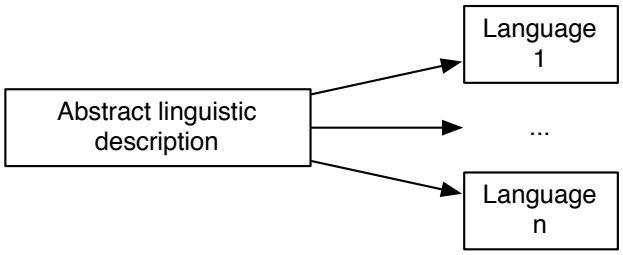

Figure 2. Multilingual grammars

The separation between abstract and concrete syntax was suggested for linguistics in (Curry 1961), using the terms "tectogrammatical" and "phenogrammatical" structure. Since the distinction has not been systematically exploited in many well-known grammar formalisms, let us summarize its main advantages.

Higher-level language descriptions The grammar writer has a greater freedom in describing the syntax for a language. As illustrated in Figure 1, when describing the abstract syntax he/she can choose not to take certain language specific details into account, such as inflection and word order. Abstracting away smaller details can make the grammars simpler, both to read and understand, and to create and maintain.

Multilingual grammar writing It is possible to define several different concrete syntax mappings for one particular abstract syntax. The abstract syntax could e.g. give a high-level description of a family of similar languages, and each concrete mapping gives a specific language instance, as shown in Figure 2.
This kind of multilingual grammar can be used as a model for interlingual translation between languages. But we do not have to restrict ourselves to only multilingual grammars; different concrete syntaxes can be given for different modalities. As an example, consider a grammar for displaying time table information. We can have one concrete syntax for writing the information as plain text, but we could also present the information in the form of a table output as a LaTeX file or in Excel format, and a third possibility is to output the information in a format suitable for speech synthesis.

Several descriptional levels Having only two descriptional levels is not a restriction; this can be generalized to as many levels as is wanted, by equating the concrete syntax of one grammar level with the abstract syntax of another

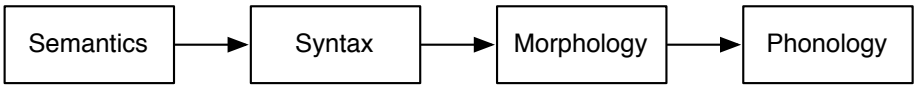

Figure 3. Several descriptional levels

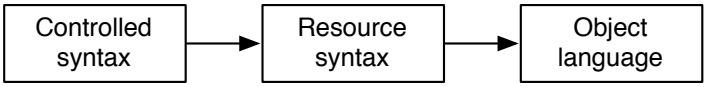

Figure 4. Using resource grammars

level. As an example we could have a spoken dialogue system with a semantical, a syntactical, a morphological and a phonological level. As illustrated in Figure 3, this system has to define three mappings; *i*) a mapping from semantical descriptions to syntax trees; *ii*) a mapping from syntax trees to sequences of lexical tokens; and *iii*) a mapping from lexical tokens to lists of phonemes.

This formulation makes grammars similar to transducers (Karttunen et al. 1996; Mohri 1997) which are mostly used in morphological analysis, but have been generalized to dialogue systems by (Lager and Kronlid 2004).

Grammar composition A multi-level grammar as described above can be viewed as a "black box", where the intermediate levels are unknown to the user. Then we are back in our first view as a grammar specifying an abstract and a concrete level together with a mapping. In this way we can talk about *grammar composition*, where the composition $G_2 \circ G_1$ of two grammars is possible if the abstract syntax of G_2 is equal to the concrete syntax of G_1.

If the grammar formalism supports this, a composition of several grammars can be pre-compiled into a compact and efficient grammar which doesn't have to mention the intermediate domains and structures. This is the case for e.g. finite state transducers, but also for GF as has been shown by Ranta (2007).

Resource grammars The possibility of separate compilation of grammar compositions opens up for writing *resource grammars* (Ranta 2007). A resource grammar is a fairly complete linguistic description of a specific language. Many applications do not need the full power of a language, but instead want to use a more well-behaved subset, which is often called a *controlled language*. Now, if we already have a resource grammar, we do not even have to write a concrete syntax for the desired controlled language, but instead we can specify the language by mapping structures in the controlled language into structures in the resource grammar, as shown in Figure 4.

3 Extending multilinguality to multimodality

Parallel multimodality *Parallel multimodality* is a straightforward instance of multilinguality. It means that the concrete syntaxes associated with an abstract syntax are not just different natural languages, but different representation modalities, encoded by language-like notations such as graphic representation formalisms. An example of parallel multimodality is given below when a route is described, in parallel, by speech and by a line drawn on a map. Both descriptions convey the full information alone, without support from the other.

This raises the dialogue management issue of whether all information should be presented in all modalities. For example, in the implementation described below all stops are indicated on the graphical presentation of a route whereas in the natural language presentation only stops where the user must change are presented. Because GF permits the suppression of information in concrete syntax, this issue can be treated on the level of grammar instead of dialogue management.

Integrated multimodality *Integrated multimodality* means that one concrete syntax representation is a combination of modalities. For instance, the spoken utterance *"I want to go from here to here"* can be combined with two pointing gestures corresponding to the two *"here"*s. It is the two modalities in combination that convey the full information: the utterance alone or the clicks alone are not enough.

How to define integrated multimodality with a grammar is less obvious than parallel multimodality. In brief, different modality "channels" are stored in different fields of a record, and it is the combination of the different fields that is sent to the dialogue system parser.

4 Proof-of-concept implementation

We have implemented a multimodal route planning system for public transport networks. The example system uses the Göteborg tram/bus network, but it can easily be adapted to other networks. User input is handled by a grammar with integrated speech and map click modalities. The system uses a grammar with parallel speech and map drawing modalities. The user and system grammars are split up into a number of modules in order to simplify reuse and modification.

The system is also multilingual, and can be used in both English and Swedish. For every English concrete syntax module shown below, there is a corresponding Swedish module. The system answers in the same language as the user made the query in.

In addition to the grammars shown below, the application consists of a number of agents which communicate using OAA (Martin et al. 1999). The grammars are used by the Embedded GF Interpreter (Bringert 2005) to parse user input and generate system output.

4.1 Transport network

The transport network is represented by a set of modules which are used in both the query and answer grammars. Since the transport network is described in a separate set of modules, the Göteborg transport network may be replaced easily. We use **cat** judgements to declare categories in the abstract syntax.

```
abstract Transport = {
  cat Stop;
}
```

The Göteborg transport network grammar extends the generic grammar with constructors for the stops. Constructors for abstract syntax terms are declared using **fun** judgements.

```
abstract Gbg = Transport ** {
  fun Angered              : Stop;
      AxelDahlstromsTorg   : Stop;
      Bergsjon             : Stop;
      ...
}
```

4.2 Multimodal input

User input is done with integrated speech and click modalities. The user may use speech only, or speech combined with clicks on the map. Clicks are expected when the user makes a query containing *"here"*.

Common declarations The QueryBase module contains declarations common to all input modalities. The Query category is used to represent the sequentialization of the multimodal input into a single value. The Input category contains the actual user queries, which will have multimodal representations. The Click category is also declared here, since it is used by both the click modality and the speech modality, as shown below.

```
abstract QueryBase = {
  cat Query;
      Input;
      Click;
  fun QInput : Input → Query;
}
```

Since QueryBase is language neutral and common to the different modalities, it has a single concrete syntax. In a concrete module, **lincat** judgements are used to declare the linearization type of a category, i.e. the type of the concrete representations of values in the category. Note that different categories may have

different linearization types. The concrete representation of abstract syntax terms is declared by **lin** judgements for each constructor in the abstract syntax.

Values in the Input category, which are intended to be multimodal, have records with one field per modality as their concrete representation. The *s1* field contains the speech input, and the *s2* field contains the click input. Terms constructed using the QInput constructor, that is sequentialized multimodal queries, are represented as the concatenation of the representations of the individual modalities, separated by a semicolon.

```
concrete QueryBaseCnc of QueryBase = {
  lincat Query = {s : Str};
         Input = {s1 : Str; s2 : Str};
         Click = {s : Str};
  lin QInput i = {s = i.s1 ++ ";" ++ i.s2};
}
```

Click modality Click terms contain a list of stops that the click might refer to:

```
abstract Click = QueryBase ** {
  cat StopList;
  fun CStops     : StopList → Click;
      NoStop     : StopList;
      OneStop    : String → StopList;
      ManyStops : String → StopList → StopList;
}
```

The same concrete syntax is used for clicks in all languages:

```
concrete ClickCnc of Click = QueryBaseCnc ** {
  lincat StopList = {s : Str};
  lin CStops xs        = {s = "[" ++ xs.s ++ "]"};
      NoStop           = {s = ""};
      OneStop x        = {s = x.s};
      ManyStops x xs = {s = x.s ++ "," ++ xs.s};
}
```

Speech modality The Query module adds basic user queries and a way to use a click to indicate a place.

```
abstract Query = QueryBase ** {
  cat Place;
  fun GoFromTo : Place → Place → Input;
      GoToFrom : Place → Place → Input;
```

```
      PClick     : Click → Place;
}
```

This module has a concrete syntax using English speech. Like terms in the Query category, Place terms are linearized to records with two fields, one for each modality.

```
concrete QueryEng of Query = QueryBaseCnc ** {
  lincat Place = {s1 : Str; s2 : Str};
  lin GoFromTo x y = {
        s1 = ["i want to go from"] ++ x.s1 ++ "to" ++ y.s1;
        s2 = x.s2 ++ y.s2
      };
      GoToFrom x y = {
        s1 = ["i want to go to"] ++ x.s1 ++ "from" ++ y.s1;
        s2 = x.s2 ++ y.s2
      };
      PClick c = {s1 = "here"; s2 = c.s};
}
```

Indexicality To refer to her current location, the user can use *"here"* without a click, or omit either origin or destination. The system is assumed to know where the user is located. Since *"here"* may be used with or without a click, inputs with two occurrences of *"here"* and only one click are ambiguous. A query might also be ambiguous even if it can be parsed unambiguously, since one click can correspond to multiple stops when the stops are close to each other on the map.

These are the abstract syntax declarations for this feature (in the Query module):

```
fun PHere     : Place;
    ComeFrom : Place → Input;
    GoTo     : Place → Input;
```

The English concrete syntax for this is added to the QueryEng module. Note that the click (*s2*) field of the linearization of an indexical *"here"* is empty, since there is no click.

```
lin PHere = {s1 = "here"; s2 = []};
    ComeFrom x = {
      s1 = ["i want to come from"] ++ x.s1;
      s2 = x.s2
    };
    GoTo x = {
      s1 = ["i want to go to"] ++ x.s1;
      s2 = x.s2
    };
```

Tying it all together The TransportQuery module ties together the transport network, speech modality and click modality modules.

```
abstract TransportQuery = Transport, Query, Click ** {
  fun PStop : Stop → Place;
}
```

4.3 Multimodal output

The system's answers to the user's queries are presented with speech and drawings on the map. This is an example of parallel multimodality as the speech and the map drawings are independent. The information presented in the two modalities is however not identical, as the spoken output only contains information about where to change trams/buses. The map output shows the entire path, including intermediate stops.

Abstract syntax for routes The abstract syntax for answers (routes) contains the information needed by all the concrete syntaxes. All concrete syntaxes might not use all of the information. A route is a non-empty list of legs, and a leg consists of a line and a list of at least two stops.

```
abstract Route = Transport ** {
  cat Route;
      Leg;
      Line;
      Stops;
  fun Then      : Leg → Route → Route;
      OneLeg    : Leg → Route;
      LineLeg   : Line → Stops → Leg;
      NamedLine : String → Line;
      ConsStop  : Stop → Stops → Stops;
      TwoStops  : Stop → Stop → Stops;
}
```

Concrete syntax for drawing routes The map drawing language contains sequences of labeled edges to be drawn on the map. The string

> drawEdge (6, [Chalmers, Vasaplatsen]); drawEdge (2, [Vasaplatsen, Gronsakstorget, Brunnsparken]);

is an example of the map drawing language described by the RouteMap concrete syntax. The TransportLabels module extended by this module is a simple concrete syntax for stops.

```
concrete RouteMap of Route = TransportLabels ** {
  lincat Route, Leg, Line, Stops = {s : Str};
```

```
  lin Then l r      = {s = l.s ++ ";" ++ r.s};
      OneLeg l      = {s = l.s ++ ";"};
      LineLeg l ss  =
        {s = "drawEdge" ++ "(" ++ l.s ++ "," ++ "[" ++ ss.s ++ "]" ++ ")"};
      NamedLine n   = {s = n.s};
      ConsStop s ss = {s = s.s ++ "," ++ ss.s};
      TwoStops x y  = {s = x.s ++ "," ++ y.s};
}
```

English concrete syntax for routes In the English concrete syntax we wish to list only the first and last stops of each leg of the route. The TransportNames module gives English representations of the stop names by replacing all non-English letters with the corresponding English ones in order to give the speech recognizer a fair chance.

```
concrete RouteEng of Route = TransportNames ** {
  lincat Route, Leg, Line = {s : Str};
         Stops = {start : Str; end : Str};
  lin Then l r      = {s = l.s ++ "." ++ r.s};
      OneLeg l      = {s = l.s ++ "."};
      LineLeg l ss  =
        {s = "Take" ++ l.s ++ "from" ++ ss.start ++ "to" ++ ss.end};
      NamedLine n   = {s = n.s};
      ConsStop s ss = {start = s.s; end = ss.end};
      TwoStops s1 s2 = {start = s1.s; end = s2.s};
}
```

5 Related Work

Johnston (1998) describes an approach to multimodal parsing where chart parsing is extended to multiple dimensions and unification is used to integrate information from different modalities. The approach described in this paper achieves a similar result by using records along with the existing unification mechanism for resolving discontinuous constituents. The main advantages of our approach are that it supports both parsing and generation, and that it does not require extending the existing formalism.

6 Conclusion

GF provides a solution to the problems named in the introduction to this paper. Abstract syntax can be used to characterise the linguistic functionality of a system in an abstract language and modality independent way. The system forces the programmer to define concrete syntaxes which completely cover the

abstract syntax. In this way, the system forces the programmer to keep all the concrete syntaxes in sync. In addition, since GF is oriented towards creating grammars from other grammars, our philosophy is that it should not be necessary for a grammar writer to have to create by hand any equivalent grammars in different formats. For example, if the grammar for the speech recogniser is to be the same as that used for interaction with dialogue management but the grammars are needed in different formats, then there should be a compiler which takes the grammar from one format to the other. Thus, for example, we have a compiler which converts a GF grammar to Nuance's format for speech recognition grammars. The idea of generating context-free speech recognition grammars from grammars in a higher-level formalism has been described by Dowding et al. (2001), and implemented in the Regulus system (Rayner et al. 2003).

Another reason for using GF grammars has to do with the use of resource grammars and cascades of levels of representation as described in section 2. This allows for the hiding of grammatical detail from language and the precise implementation of modal interaction for other modalities. This enables the dialogue system developer to reuse previous grammar or modal interaction implementations without herself having to reprogram the details for each new dialogue system. Thus the dialogue engineer need not be a grammar engineer or an expert in multimodal interfaces.

References

Björn Bringert. Embedded Grammars. Master's thesis, Chalmers University of Technology, Göteborg, Sweden, February 2005. URL `http://www.cs.chalmers.se/~bringert/publ/exjobb/embedded-grammars.pdf`.

Coq. *The Coq Proof Assistant Reference Manual*. The Coq Development Team, 1999. Available at `http://pauillac.inria.fr/coq/`

Haskell B. Curry. Some Logical Aspects of Grammatical Structure. In Roman O. Jakobson, editor, *Structure of Language and its Mathematical Aspects, volume 12 of Symposia on Applied Mathematics*, pages 56–68. American Mathematical Society, Providence, 1961.

John Dowding, Beth A. Hockey, Jean M. Gawron, and Christopher Culy. Practical issues in compiling typed unification grammars for speech recognition. In *ACL '01: Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pages 164–171, Morristown, NJ, USA, 2001. Association for Computational Linguistics. doi: 10.3115/1073012.1073034.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993. ISSN 0004-5411. doi: 10.1145/138027.138060.

Michael Johnston. Unification-based multimodal parsing. In *Proceedings of the 36th annual meeting on Association for Computational Linguistics*, pages 624–630, Morristown, NJ, USA, 1998. Association for Computational Linguistics. URL `http://portal.acm.org/citation.cfm?id=980949`.

Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328, 1996.

Torbjörn Lager and Fredrik Kronlid. The Current platform: Building conversational agents in Oz. In *2nd International Mozart/Oz Conference*, October 2004.

Lena Magnusson and Bengt Nordström. *The Alf proof editor and its proof engine*, volume 806 of *Lecture Notes in Computer Science*, pages 213–237. Springer, 1994. doi: 10.1007/3-540-58085-9_78.

David L. Martin, Adam J. Cheyer, and Douglas B. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1-2):91–128, 1999. doi: 10.1080/088395199117504.

Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, May 1997. ISBN 0262631814.

Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–312, 1997.

Simon Peyton Jones. The Haskell 98 Language. *Journal of Functional Programming*, 13(1):1–146, 2003.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004. ISSN 0956-7968. doi: 10.1017/S0956796803004738.

Aarne Ranta. Modular Grammar Engineering in GF. *Research on Language and Computation*, 5(2):133–158, June 2007. doi: 10.1007/s11168-007-9030-6.

Manny Rayner, Beth A. Hockey, and John Dowding. An open source environment for compiling typed unification grammars into speech recognisers. In *EACL '03: Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics*, pages 223–226, Morristown, NJ, USA, 2003. Association for Computational Linguistics. ISBN 1111567890. URL `http://portal.acm.org/citation.cfm?id=1067790`.

Paper III | Rapid Development of Dialogue Systems by Grammar Compilation

SIGDIAL 2007, Antwerp (Extended version)

Rapid Development of Dialogue Systems by Grammar Compilation

Björn Bringert
Department of Computer Science and Engineering
Chalmers University of Technology
and University of Gothenburg
SE-412 96 Göteborg, Sweden
bringert@chalmers.se

Abstract

We propose a method for rapid development of dialogue systems where a Grammatical Framework (GF) grammar is compiled into a complete VoiceXML application. This makes dialogue systems easy to develop, maintain, localize, and port to other platforms, and can improve the linguistic quality of generated system output. We have developed compilers which produce VoiceXML dialogue managers and JavaScript linearization code from GF grammars. Along with the existing GF speech recognition grammar compiler, this makes it possible to produce a complete mixed-initiative information-seeking dialogue system from a single GF grammar.

1 Introduction

In current industrial practice, dialogue systems are often constructed using VoiceXML for dialogue management, context-free speech recognition grammars for input, with semantic tags for interpretation, and concatenation of canned text and output data for system responses. Developing several components which all need to cover the same concepts increases development costs. Having multiple interdependent components in formalisms with few automatic correctness and consistency checks also complicates maintenance, since any change in the coverage of one component may require changes in the others. Since all the components are language-specific, much effort is needed to port the system to a new language, and to keep the implementations for different languages in sync. The lack of a powerful method for output realization makes it hard to generate high-quality output, especially for languages with a more complex morphology than English.

This is analogous to the developments in programming languages. Computer machine languages are designed to be easy to implement in hardware, but

it is tedious for humans to write programs in machine or assembly code. One approach to solving this problem was to add more and more powerful instructions to the assembly and machine languages. However, there is a limit to how complex a machine language can be, and there are many inherent limitations in machine languages. A partly parallel, and much more successful, trend has been the development of high-level programming languages, which are translated automatically by a compiler to the low-level languages. Thus, we have a division of programming languages into low-level languages, which are close to the concrete workings of the machine, and high-level languages, which are easier for humans to write programs in. A high-level programming language can be compiled to several different low-level languages, for example machine code for different machine architectures. They may also be interpreted rather than compiled.

We propose that the formalisms used for the construction of dialogue systems, and other related natural language applications, could benefit from a similar development. We specify systems in a single high-level formalism, which is then compiled into the existing lower-level formalisms. The developer writes a GF abstract syntax module which defines the user input and system output semantics, and a concrete syntax module which describes how each construct in the semantics is represented in natural language. The GF grammar is then compiled to a complete VoiceXML application. The dialogue flow is determined by the abstract syntax (ontology) of the grammar. This is based on the idea by Ranta and Cooper (2004) that a proof editor for constructive type theory can be used to implement the information gathering phase of information-seeking dialogue systems. Figure 1 illustrates the compilation. In addition to the generated components, the developer writes JavaScript code to connect the dialogue system to any domain-specific resources, such as databases or other external systems.

We start by giving an introduction to the GF formalism. In the following section we show how to write a GF grammar for a small example dialogue system, and outline how this system can be extended. We then describe the compilation process in some more detail. We conclude by presenting some possible future extensions to our compilers, and related work on dialogue system engineering.

2 Grammatical Framework

Grammatical Framework (GF) (Ranta 2004) is a grammar formalism based on constructive type theory. GF separates grammar into *abstract syntax* and *concrete syntax*, corresponding to Curry's (1961) division of grammar into *tectogrammar* and *phenogrammar*.

2.1 Abstract Syntax

The abstract syntax defines the ontology of the application, that is, *what* can be said. An abstract syntax contains category (**cat**) and function (**fun**) definitions.

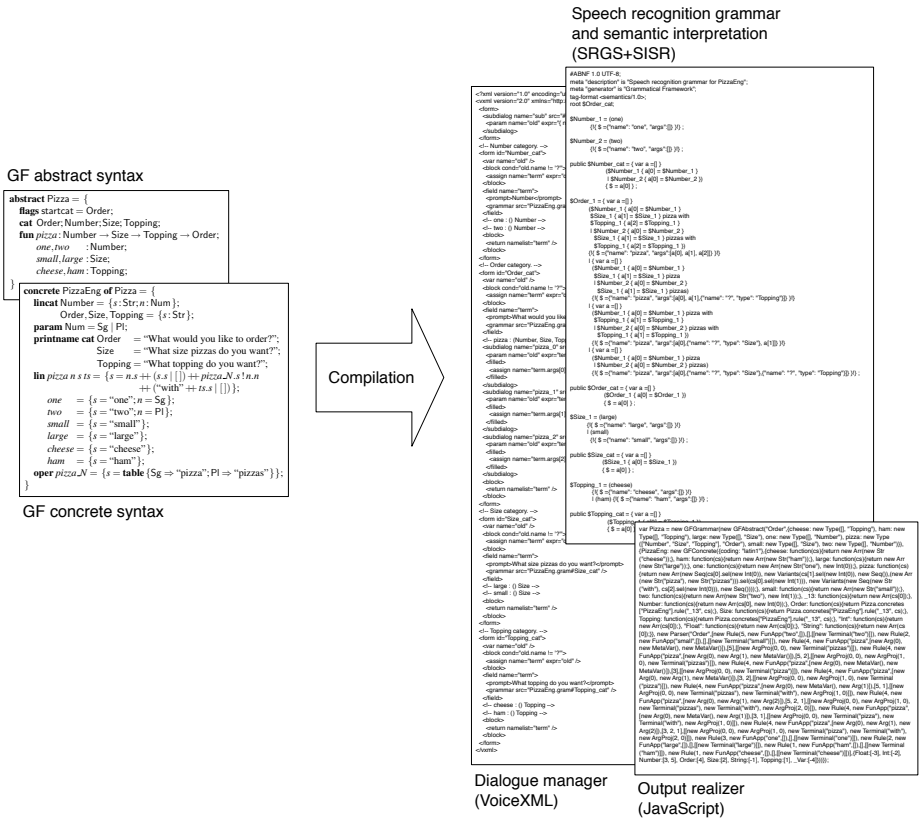

Figure 1. Dialogue system generation.

This is an example of a small abstract syntax:

cat Order; Size;
fun *pizza* : Size → Order;
small : Size;

This allows us to construct an abstract syntax term *pizza small* of type Order. In addition to functions, abstract syntax terms can also contain *metavariables*, written ?. For example, the term *pizza* ? contains a metavariable of type Size. As we will see later, metavariables are essential to our dialogue management implementation.

2.2 Concrete Syntax

A concrete syntax defines *how* each abstract syntax construct is realized in a particular language. From a concrete syntax, the GF system can derive both parsing and realization components. A concrete syntax contains linearization

type (**lincat**) and linearization (**lin**) definitions. The linearization type of a category is the type of the concrete syntax terms produced for abstract syntax terms in the given category. A linearization definition is a function from the linearizations of the arguments of an abstract syntax term to a concrete syntax term. Terms in concrete syntax can be records, strings, tables, and parameters. This is an example of a concrete syntax for the abstract syntax above:

lincat Order, Size = $\{s : \mathsf{Str}\}$;
lin *pizza* $x = \{s =$ "a" $++ x.s ++$ "pizza"$\}$;
small $= \{s =$ "small"$\}$;

3 An Example Dialogue System

A spoken language dialogue system requires at least the following components: a *speech recognizer*, guided by a *language model*, some form of *semantic interpretation* component, *output realization*, *speech synthesis*, *dialogue management* and *domain resources* such as databases or other external systems.

Of these components, speech recognizers and speech synthesizers for many languages are available as commercial off-the-shelf products. The domain resources are normally domain-specific and somewhat outside the dialogue system itself. The remaining components: language model, semantic interpretation, output realization, and dialogue management are often developed as separate components. This leads to duplicated effort in developing many components which all need to cover the same concepts. Having multiple interdependent components also complicates maintenance, since any change in the coverage of one component may require changes in the others. Furthermore, the low-level formalisms often have little automatic correctness and consistency checking. The resulting components are also often language specific, which makes makes porting to new languages difficult.

This section shows a GF grammar from which a complete dialogue system (excluding the domain resources) can be derived automatically. For reasons of brevity, this system is very small. An extended version of this system is available online[1].

3.1 Abstract Syntax

The abstract syntax in Figure 2 describes the possible things that the user can say, in a semantic form. There is one category for each kind of input. In this application, the main input object is an Order. An order can in this small example only be for a number of pizzas, all of the same size and with the same topping. A number is "one" or "two", the sizes are "small" and "large", and the toppings are "ham" and "cheese". An example abstract syntax term in the Order category is: *pizza two small cheese*.

[1] `http://www.cs.chalmers.se/~bringert/xv/pizza/`

```
abstract Pizza = {
  flags startcat = Order;
  cat Order; Number; Size; Topping;
  fun pizza : Number → Size → Topping → Order;
      one, two     : Number;
      small, large : Size;
      cheese, ham  : Topping;
}
```

Figure 2. Abstract syntax for the example system.

3.2 Concrete Syntax

The concrete syntax in Figure 3 defines how the terms in the abstract syntax are realized (and inversely, how concrete syntax terms can be interpreted as representations of abstract syntax terms). For example, the linearization type of Topping is $\{s : \mathsf{Str}\}$, that is, a record with a single field s which contains a string. The linearization for *cheese* is the concrete syntax term $\{s =$ "cheese"$\}$. When parsing and realizing, the s field is used as the textual representation of the term, as it would be input by a user, or output by the system.

The linearization type of Number contains a field n, which is used for agreement. The type of n is Num, defined by a **param** definition to be either Sg or Pl. In the linearization of *pizza*, the n field of the Number is used to inflect the noun "pizza".

An important feature of this grammar is that it allows partially specified input. While the utterance "two small pizzas with cheese" results in the abstract syntax term *pizza two small cheese*, the partial versions "two pizzas with cheese" (*pizza two* ? *cheese*), "two small pizzas" (*pizza two small* ?), and "two pizzas" (*pizza two* ? ?) are also allowed. The intention is that the system will ask follow-up questions to replace all metavariables with complete terms. This process is type-directed: the system asks for a subterm of the appropriate type. The user can give complete input, or some parts can be omitted. The system will then prompt for any omitted parts. Partial input, implemented with suppression, is thus used to achieve a mixed-initiative dialogue. An interesting special case is when no suppression is used in the grammar. This gives a user-driven dialogue, where the system will not ask follow-up questions.

The **printname** definitions are used as prompts for each category.

3.3 Example Dialogues

The system generated from the grammar in the previous section allows dialogues such as the examples below. After each user action we show the information state, i.e. the current state of the abstract syntax term that we are constructing.

S: What would you like to order?

```
concrete PizzaEng of Pizza = {
  lincat Number = {s : Str; n : Num};
         Order, Size, Topping = {s : Str};
  param Num = Sg | Pl;
  printname cat Order   = "What would you like to order?";
                Size    = "What size pizzas do you want?";
                Topping = "What topping do you want?";
  lin pizza n s ts = {s = n.s ++ (s.s | []) ++ pizza_N.s ! n.n
                          ++ ("with" ++ ts.s | [])};
      one    = {s = "one"; n = Sg};
      two    = {s = "two"; n = Pl};
      small  = {s = "small"};
      large  = {s = "large"};
      cheese = {s = "cheese"};
      ham    = {s = "ham"};
  oper pizza_N = {s = table {Sg => "pizza"; Pl => "pizzas"}};
}
```

Figure 3. Concrete syntax for the example system.

U: two pizzas
pizza two ? ?
S: What size pizzas do you want?
U: small
pizza two small ?
S: What topping do you want?
U: ham
pizza two small ham

Here, more information is given in the first answer:

S: What would you like to order?
U: two pizzas with ham
pizza two ? *ham*
S: What size pizzas do you want?
U: small
pizza two small ham

3.4 Extending the Example System

Recursive structures One possible extension to the example system is to use a recursive structure to allow more complex orders:

```
cat Order; Item; [Item];
fun order : [Item] → Order;
    pizza : Number → Size → Topping → Item;
```

```
printname cat [Item] = "Anything else?";
lin order is          = {s = is.s};
    ConsItem x xs = {s = x.s ++ ("and" ++ xs.s | [])};
    BaseItem      = {s = "nothing" ++ "else"};
```

Here, $BaseItem : [\mathsf{Item}]$ and $ConsItem : \mathsf{Item} \to [\mathsf{Item}] \to [\mathsf{Item}]$ are the two functions in the new category [Item]. While recursive structures can be implemented with subdialogues and scripting in VoiceXML (by essentially writing by hand the code that we generate), it appears to be beyond the scope of standard practice. If we also add drinks as a kind of Item, the system will support dialogues such as this one:

S: What would you like to order?
U: one large pizza
order [*pizza one large* ?, ?]
S: What topping would you like?
U: cheese
order [*pizza one large cheese*, ?]
S: Anything else?
U: one beer
order [*pizza one large cheese*, *drink one beer*, ?]
S: Anything else?
U: nothing else
order [*pizza one large cheese*, *drink one beer*]

System output At the end of the dialogue, we would like the system to give a response based on the output of some domain resource. For example, the pizza ordering system might return the price of the order. This could be used to construct a confirmation using an addition to the grammar:

```
cat Output;
fun confirm : Order → Number → Output;
lin confirm o p = {s = o.s ++ "costs" ++ p.s ++ "euros"}
```

Multilinguality To port a dialogue system to a new language, all that needs to be done is to write a new concrete syntax. For many languages, writing speech recognition grammars and realization functions is more complicated than for English. For example, Swedish adjectives agree with the gender and number of the noun they modify. GF's expressive concrete syntax makes it possible to implement such features with little effort, and if the GF Resource Grammar Library is used, it is as easy to write the Swedish grammar as the English.

Multimodality GF can be used to write multimodal grammars (Bringert et al. 2005). The extended online version (see screenshot in Figure 4) of the example system uses a concrete syntax which linearizes pizza and drink orders to vector drawings to display graphical representations of the completed orders.

Figure 4. A screenshot of a multimodal XHTML+Voice dialogue system generated from an extended version of the grammar shown above.

4 Implementation

This section describes how the different dialogue system components are generated from a GF grammar.

4.1 Dialogue Management

We have implemented a compiler which produces VoiceXML code which implements the kind of dialogue management shown in the examples above. The input to the compiler is the abstract syntax definition and the prompts defined in the concrete syntax.

The generated VoiceXML document contains one form for each category in the GF grammar. Each form takes an argument, which the caller sets to the currently known abstract syntax term. In the top-level call this argument is ?. If the given term is a metavariable, speech input is requested in the speech recognition grammar category that corresponds to the GF category that the form is for. Function arguments which are not specified in the recognized phrase result in metavariables in the abstract syntax tree returned by the semantic interpretation code. The form then traverses the abstract syntax tree, and refines any metavariables by subdialogue calls to the forms for the appropriate categories.

Dialogue systems built using our approach can be run in standard VoiceXML

interpreters, and in web browsers which support XHTML+Voice (Axelsson et al. 2004). Such web browsers are currently available for both personal and handheld computers. This means that all that is needed to very rapidly develop and deploy a relatively sophisticated dialogue system to a very large number of users is the GF grammar compiler and a web server.

4.2 Language Model and Semantic Interpretation

The GF grammar is also compiled (Bringert 2007) to a speech recognition grammar in JSGF or SRGS format, with embedded semantic interpretation code in SISR format. The generated grammar has one top-level category for each category in the GF abstract syntax. The embedded semantic information constructs abstract syntax terms represented as JavaScript objects.

4.3 Generation

GF's linearization rules are essentially a simple functional programming language which can be interpreted directly, compiled to a simpler language and interpreted, or compiled to code in some other programming language. We have implemented a compiler which translates GF linearization rules to JavaScript code. The generated code be used with VoiceXML implementations, since they include JavaScript interpreters, or as a stand-alone component in web pages. The generated code implements a linearization function which produces a string from an abstract syntax term.

This makes it possible to generate output of a high linguistic quality in VoiceXML applications, something which is difficult to achieve using hand-written VoiceXML code.

5 Future Work

5.1 Dialogue flexibility

We have explained how to implement simple task-oriented mixed-initiative dialogues. However, there are still many possibilities for more flexible dialogue left to be investigated. It would be interesting to see, for example, what additional items from the Trindi tick list (Bohlin et al. 1999) can be elegantly implemented by using GF grammars.

5.2 Automatically Generated Help

When the user asks for help in a given form, or if the user's input cannot be recognized, an example utterance in the corresponding category could be output. For each category, GF could produce an example utterance by linearizing a randomly generated abstract syntax term of the right type. The random generation may have to be guided to produce examples that are not too simple or too complex. A more sophisticated approach would be to use a statistical

language model to make a guess at what the user wanted to say, and let that guide the generation of the example utterance. Hockey et al. (2003) have found such feedback to significantly improve the performance of dialogue systems for naive users.

5.3 Context-dependent Prompts

In the approach described here, there can only be one system question for each category. This means that if the same category is used in multiple positions, extra categories may have to be added. Perhaps it could be possible to have the prompt depend on what argument position it is for, rather than just on the category. However, the prompt sometimes need to depend not only on what the result is used for, but also on the already given information. For example, in the pizza system, we would like the prompt for size to depend on the number of pizzas ordered, instead of the current:

U: I would like one pizza with ham
S: What size pizzas do you want?

5.4 Dependent Types

As Ranta and Cooper (2004) have shown, dependently typed abstract syntax can be used to guide refinement with more precision. Support for generating speech-recognition grammar from dependently typed grammars has been implemented. However, this only supports grammars where the dependencies are on finite types, and the generated VoiceXML and semantic interpretation code does not yet support dependent types.

5.5 Integrated Multimodality

In addition to the *parallel multimodality* described in Section 3.4, where the complete information is represented independently in multiple modalities, GF grammars (Bringert et al. 2005) can also be used for *integrated multimodality*, where information in multiple modalities together convey meaning. The fact that we use semantic information embedded in the speech recognition grammar to build abstract syntax trees makes it difficult to handle integrated multimodality with GF grammars. The recently developed JavaScript GF parser (Meza Moreno and Bringert 2008) could be used to instead perform parsing as a separate step, which would allow integrated multimodality.

5.6 Weighted Grammars

GF grammars can be annotated with weights for each function in the abstract syntax. These weights could be carried over to the generated speech recognition grammars. Rayner et al. (2006) found that weighted rules can significantly improve recognition performance for generated grammars.

6 Related Work

6.1 Dialogue and Proof Editing

Ranta and Cooper (2004) introduced the idea of using a proof editor for constructive type theory as a dialogue system. Our dialogue manager generation implements this idea by compiling GF abstract syntax to VoiceXML, rather than the more direct interpretation outlined in the original article.

6.2 GUI Tools for Rapid Dialogue System Development

The CSLU Rapid Application Developer (McTear 1999) is a toolkit for rapid development of dialogue systems. Systems are built using a graphical interface with a focus on the dialogue flow.

The Application Generation Platform (AGP) (Hamerich et al. 2004) developed in the GEMINI project can generate multilingual and multimodal interfaces to existing databases semi-automatically. The developer is guided through the database-centric process by GUI assistants.

In contrast to these toolkits, we use a compiler-like model, rather than a graphical design environment. In addition, our development model is focused on the specification and realization of the inputs and outputs of the system, rather than on the dialogue flow or the underlying database.

6.3 GF and Dialogue Systems

GF has already been used in the implementation of a number of dialogue systems (Ericsson et al. 2006). Those systems make use of the same speech recognition grammar compiler as in the approach described here. However, where we based the dialogue management on the structure of the GF grammar, they rely on an external dialogue manager with dialogue plans separate from the grammar.

7 Conclusions

The distinction between abstract and concrete syntax is well established in the field of compiler technology, but not within computational linguistics. There are now a number of type theoretic grammar formalisms, including Abstract Categorial Grammar (de Groote 2001), Lambda Grammar (Muskens 2001), Higher Order Grammar (Pollard 2004) and Grammatical Framework (Ranta 2004). Our work demonstrates a practical application of type theoretic grammars, and exploits the abstract syntax–concrete syntax dichotomy in an essential way.

The GF source for the basic version of the example application shown above consists of 24 lines (1065 characters). From this, our compilers generate three different components, which together contain 151 lines (7690 characters) of code.

We have shown that GF grammars can be used to implement mixed-initiative information-seeking dialogue systems. From the declarative and linguistically

powerful specification that a GF grammar is, we generate the interconnected components needed to run dialogue systems using industry standard infrastructure. Hopefully, this method can reduce the development and maintenance costs for dialogue systems, and at the same time improve their linguistic quality. The methods described in this paper are implemented as part of the open source GF system[2].

Acknowledgments

While any errors in the article are solely the responsibility of the author, we would like to thank Aarne Ranta, Håkan Burden, and Robin Cooper for comments on this work. We would also like to thank OptimSys, s.r.o., for providing us with their OptimTalk VoiceXML interpreter, and for their support in making sure that it runs our generated code. The developers at Opera Software ASA and ROBO Design have helped with XHTML+Voice issues. The code in this paper has been typeset using `lhs2TeX`, with help from Andres Löh. This work has been partly funded by the EU TALK project, IST-507802, and Library-Based Grammar Engineering, Swedish Research Council project dnr 2005-4211.

References

Jonny Axelsson, Chris Cross, Jim Ferrans, Gerald McCobb, T. V. Raman, and Les Wilson. XHTML+Voice profile 1.2. Specification, VoiceXML Forum, 2004. URL `http://www.voicexml.org/specs/multimodal/x+v/12/`.

Peter Bohlin, Johan Bos, Staffan Larsson, Ian Lewin, Colin Matheson, and David Milward. Survey of Existing Interactive Systems. Deliverable 1.3, TRINDI, 1999. URL `http://www.ling.gu.se/~peb/pubs/BohlinBosLarsson-1999a.pdf`.

Björn Bringert. Speech Recognition Grammar Compilation in Grammatical Framework. In *Proceedings of the Workshop on Grammar-Based Approaches to Spoken Language Processing, Prague, Czech Republic*, pages 1–8. Association for Computational Linguistics, June 2007. URL `http://www.aclweb.org/anthology/W/W07/W07-1801`.

Björn Bringert, Robin Cooper, Peter Ljunglöf, and Aarne Ranta. Multimodal Dialogue System Grammars. In *Proceedings of DIALOR'05, Ninth Workshop on the Semantics and Pragmatics of Dialogue, Nancy, France*, pages 53–60, June 2005. URL `http://dialor05.loria.fr/Papers/07-BjornBringert.pdf`.

Haskell B. Curry. Some Logical Aspects of Grammatical Structure. In Roman O. Jakobson, editor, *Structure of Language and its Mathematical Aspects,*

[2] See `http://www.digitalgrammars.com/gf/`

volume 12 of Symposia on Applied Mathematics, pages 56–68. American Mathematical Society, Providence, 1961.

Philippe de Groote. Towards abstract categorial grammars. In *Proceedings of 39th Annual Meeting of the Association for Computational Linguistics, Toulouse, France*, pages 252–259, Morristown, NJ, USA, July 2001. Association for Computational Linguistics. doi: 10.3115/1073012.1073045.

Stina Ericsson, Gabriel Amores, Björn Bringert, Håkan Burden, Ann-Charlotte Forslund, David Hjelm, Rebecca Jonson, Staffan Larsson, Peter Ljunglöf, Pilar Manchón, David Milward, Guillermo Pérez, and Mikael Sandin. Software illustrating a unified approach to multimodality and multilinguality in the in-home domain. deliverable 1.6, 2006. URL `http://www.talk-project.org/fileadmin/talk/publications_public/deliverables_public/D1_6.pdf`.

Stefan Hamerich, Volker Schubert, Volker Schless, Ricardo de Córdoba, José M. Pardo, Luis F. d'Haro, Basilis Kladis, Otilia Kocsis, and Stefan Igel. Semi-Automatic Generation of Dialogue Applications in the GEMINI Project. In Michael Strube and Candy Sidner, editors, *Proceedings of the 5th SIGdial Workshop on Discourse and Dialogue*, pages 31–34, Cambridge, Massachusetts, USA, 2004. Association for Computational Linguistics. URL `http://acl.ldc.upenn.edu/hlt-naacl2004/sigdial04/pdf/hamerich.pdf`.

Beth A. Hockey, Oliver Lemon, Ellen Campana, Laura Hiatt, Gregory Aist, James Hieronymus, Alexander Gruenstein, and John Dowding. Targeted help for spoken dialogue systems: intelligent feedback improves naive users' performance. In *EACL '03: Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics*, pages 147–154, Morristown, NJ, USA, 2003. Association for Computational Linguistics. ISBN 1333567890. doi: 10.3115/1067807.1067828.

Michael F. McTear. Software to support research and development of spoken dialogue systems. In *Proceedings, Sixth European Conference on Speech Communication and Technology (EUROSPEECH'99), Budapest, Hungary*, pages 339–342. ISCA Archive, September 1999. URL `http://www.cslu.ogi.edu/toolkit/pubs/pdf/mctear_EUROSPEECH_99.pdf`.

Moisés S. Meza Moreno and Björn Bringert. Interactive Multilingual Web Applications with Grammatical Framework. In Bengt Nordström and Aarne Ranta, editors, *Advances in Natural Language Processing, 6th International Conference, GoTAL 2008, Gothenburg, Sweden*, volume 5221 of *Lecture Notes in Computer Science*, pages 336–347, Heidelberg, August 2008. Springer. doi: 10.1007/978-3-540-85287-2_32.

Reinhard Muskens. Categorial Grammar and Lexical-Functional Grammar. In Miriam Butt and Tracy H. King, editors, *Proceedings of the LFG01 Conference, University of Hong Kong, Hong Kong*. CSLI Publications, 2001. URL `http://let.uvt.nl/general/people/rmuskens/pubs/cglfg.pdf`.

Carl Pollard. Higher-Order Categorial Grammar. In *Proceedings of Categorial Grammars 2004*, pages 340–361, June 2004. URL `http://www.ling.ohio-state.edu/~hana/hog/pollard2004-CG.pdf`.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004. ISSN 0956-7968. doi: 10.1017/S0956796803004738.

Aarne Ranta and Robin Cooper. Dialogue Systems as Proof Editors. *Journal of Logic, Language and Information*, 13(2):225–240, 2004. ISSN 0925-8531. doi: 10.1023/B:JLLI.0000024736.34644.48.

Manny Rayner, Pierrette Bouillon, Beth A. Hockey, and Nikos Chatzichrisafis. REGULUS: A Generic Multilingual Open Source Platform for Grammar-Based Speech Applications. In *Proceedings of LREC*, May 2006. URL `http://www.issco.unige.ch/pub/RaynerEA_LREC2006.pdf`.

Paper IV

Speech Translation with Grammatical Framework

Coling 2008 Workshop on Speech Processing for Safety Critical Translation and Pervasive Applications, Manchester

Speech Translation with Grammatical Framework

Björn Bringert
Department of Computer Science and Engineering
Chalmers University of Technology
and University of Gothenburg
bringert@chalmers.se

Abstract

Grammatical Framework (GF) is a grammar formalism which supports interlingua-based translation, library-based grammar engineering, and compilation to speech recognition grammars. We show how these features can be used in the construction of portable high-precision domain-specific speech translators.

1 Introduction

Speech translators for safety-critical applications such as medicine need to offer high-precision translation. One way to achieve high precision is to limit the coverage of the translator to a specific domain. The development of such high-precision domain-specific translators can be resource intensive, and require rare combinations of developer skills. For example, consider developing a Russian–Swahili speech translator for the orthopedic domain using direct translation between the two languages. Developing such a system could require an orthopedist programmer and linguist who speaks Russian and Swahili. Such people may be hard to find. Furthermore, developing translators for all pairs of N languages requires $O(N^2)$ systems, developed by an equal number of bilingual domain experts.

The language pair explosion and the need for the same person to possess knowledge about the source and target languages can be avoided by using an interlingua-based approach. The requirement that developers be both domain experts and linguists can be addressed by the use of grammar libraries which implement the domain-independent linguistic details of each language.

Grammatical Framework (GF) (Ranta 2004) is a type-theoretic grammar formalism which is well suited to high-precision domain-specific interlingua-based translation (Khegai 2006), and library-based grammar engineering (Ranta 2008). GF divides grammars into *abstract syntax* and *concrete syntax*. The abstract syntax defines *what* can be said in the grammar, and the concrete syntax defines *how* it is said in a particular language. If one abstract syntax

syntax is given multiple concrete syntaxes, the abstract syntax can be used as an interlingua. Given an abstract and a concrete syntax, GF allows both parsing (text to abstract syntax) and linearization (abstract syntax to text). This means that interlingua-based translation is just a matter of parsing in one language and linearizing to another.

The GF resource grammar library (Ranta 2008) implements the domain-independent morphological and syntactic details of eleven languages. A grammar writer can use functions from a resource grammar when defining the concrete syntax of an application grammar. This is made possible by GF's support for *grammar composition*, and frees the grammar writer from having to implement linguistic details such as agreement, word order etc.

In addition to parsing and linearization, the declarative nature of GF grammars allows them to be compiled to other grammar formats. The GF speech recognition grammar compiler (Bringert 2007) can produce context-free grammars or finite-state models which can be used to guide speech recognizers.

These components, interlingua-based translation, grammar libraries, and speech recognition grammar compilation, can be used to develop domain-specific speech translators based on GF grammars. Figure 1 shows an overview of a minimal unidirectional speech translator which uses these components. This is a proof-of-concept system that demonstrates how GF components can be used for speech translation, and as such it can hardly be compared to a more complete and mature system such as MedSLT (Bouillon et al. 2005). However, the system has some promising features compared to systems based on unification grammars: the expressive power of GF's concrete syntax allows us to use an application-specific interlingua without any transfer rules, and the wide language support of the GF Resource Grammar library makes it possible to quickly port applications to new languages.

In Section 2 we show a small example grammar for a medical speech translator. Section 3 briefly discusses how a speech translator can be implemented. Section 5 describes some possible extensions to the proof-of-concept system, and Section 6 offers some conclusions.

2 Example Grammar

We will show a fragment of a grammar for a medical speech translator. The example comes from Khegai's (2006) work on domain-specific translation with GF, and has been updated to use the current version of the GF resource library API.

The small abstract syntax (interlingua) shown in Figure 2 has three categories (**cat**): the start category Prop for complete utterances, Patient for identifying patients, and Medicine for identifying medicines. Each category contains a single function (**fun**). There are the nullary functions ShePatient and PainKiller, and the binary NeedMedicine, which takes a Patient and a Medicine as arguments, and produces a Prop. This simple abstract syntax only allows us to construct the term NeedMedicine ShePatient PainKiller. A larger version could

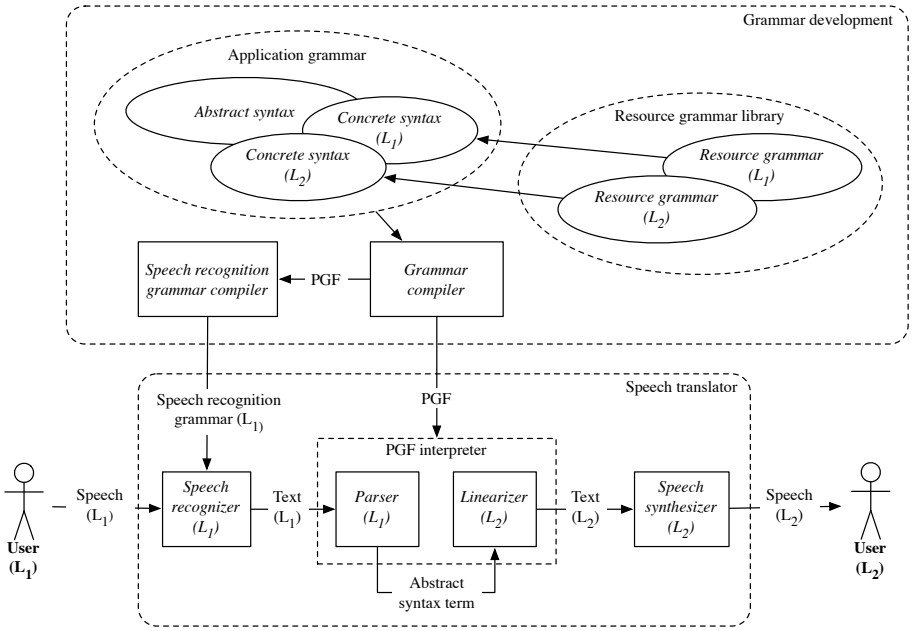

Figure 1. Overview of a GF-based speech translator. The developer writes a multilingual application grammar using the resource grammar library. This is compiled to a PGF (Portable Grammar Format) grammar used for parsing and linearization, and a speech recognition grammar. Off-the-shelf speech recognizers and speech synthesizers are used together with a PGF interpreter in the running system.

for example include categories for body parts, symptoms and illnesses, and more functions in each category. An example of a term in such an extended grammar could be And (Injured TheyPatient Foot) (NeedMedicine HePatient Laxative).

For this abstract syntax we can use the English resource grammar to write an English concrete syntax, as shown in Figure 3. The resource grammar category NP is used as the linearization type (**lincat**) of the application grammar categories Patient and Medicine, and S is used for Prop. The linearizations (**lin**) of each abstract syntax function use overloaded functions from the resource grammar, such as *mkCl* and *mkN* which create clauses and nouns, respectively.

Figure 4 shows a Swedish concrete syntax created in the same way. Note that PainKiller in Swedish uses a mass noun construction rather than the indefinite article.

3 Speech Translator Implementation

The GF grammar compiler takes grammars in the GF source language used by programmers, and produces grammars in a low-level language (Portable Gram-

```
abstract Health = {
  flags startcat = Prop;
  cat Patient;
      Medicine;
      Prop;
  fun ShePatient    : Patient;
      PainKiller    : Medicine;
      NeedMedicine : Patient → Medicine → Prop;
}
```

Figure 2. Example abstract syntax.

```
concrete HealthEng of Health = open TryEng in {
  flags language = en_US;
  lincat Patient, Medicine = NP;
         Prop              = S;
  lin ShePatient           = mkNP she_Pron;
      PainKiller           = mkNP a_Art (mkN "painkiller");
      NeedMedicine p m = mkS (mkCl p (mkV2 (mkV "need")) m);
}
```

Figure 3. English concrete syntax.

mar Format, PGF (Angelov et al. 2008)) for which interpreters can be easily and efficiently implemented. There are currently PGF implementations in Haskell, Java and JavaScript. The GF speech recognition grammar compiler (Bringert 2007) targets many different formats, including Nuance GSL, SRGS, JSGF and HTK SLF. This means that speech translators based on GF can easily be implemented on almost any platform for which there is a speech recognizer and speech synthesizer. We have run Java-based versions under Windows using Nuance Recognizer and RealSpeak or FreeTTS, Haskell-based versions under Linux using Nuance Recognizer and RealSpeak, and JavaScript-based prototypes in the Opera XHTML+Voice-enabled web browser on Zaurus PDAs and Windows desktops.

The speech translation system itself is domain-independent. All that is required to use it in a new domain is an application grammar for that domain.

4 Evaluation

Since we have presented a proof-of-concept system that demonstrates the use of GF for speech translation, rather than a complete system for any particular domain, quantitative translation performance evaluation would be out of place.

```
concrete HealthSwe of Health = open TrySwe in {
  flags coding = utf8;
        language = sv_SE;
  lincat Patient, Medicine = NP;
         Prop              = S;
  lin ShePatient           = mkNP she_Pron;
      PainKiller           = mkNP (mkN "smärtstillande");
      NeedMedicine p m = mkS (mkCl p (mkV2 (mkV "behöver")) m);
}
```

Figure 4. Swedish concrete syntax.

Rather, we have evaluated the portability and speed of prototyping. Our basic speech translators written in Java and Haskell, using existing speech components and PGF interpreters, require less than 100 lines of code each. Developing a small domain for the translator can be done in under 10 minutes.

5 Extensions

5.1 Interactive Disambiguation

The concrete syntax for the source language may be ambiguous, i.e. there may be sentences for which parsing produces multiple abstract syntax terms. The ambiguity can sometimes be preserved in the target language, if all the abstract syntax terms linearize to the same sentence.

In cases where the ambiguity cannot be preserved, or if we want to force disambiguation for safety reasons, we can use a *disambiguation grammar* to allow the user to choose an interpretation. This is a second concrete syntax which is completely unambiguous. When the user inputs an ambiguous sentence, the system linearizes each of the abstract syntax terms with the disambiguation grammar, and prompts the user to select the sentence with the intended meaning. If only some of the ambiguity can be preserved, the number of choices can be reduced by grouping the abstract syntax terms into equivalence classes based on whether they produce the same sentences in the target language. Since all terms in a class produce the same output, the user only needs to select the correct class of unambiguous sentences.

Another source of ambiguity is that two abstract syntax terms can have distinct linearizations in the source language, but identical target language linearizations. In this case, the output sentence will be ambiguous, even though the input was unambiguous. This could be addressed by using unambiguous linearizations for system output, though this may lead to the use of unnatural constructions.

5.2 Bidirectional Translation

Since GF uses the same grammar for parsing and linearization, the grammar for a translator from L_1 to L_2 can also be used in a translator from L_2 to L_1, provided that the appropriate speech components are available. Two unidirectional translators can be used as a bidirectional translator, something which is straightforwardly achieved using two computers. While PGF interpreters can already be used for bidirectional translation, a single-device bidirectional speech translator requires multiplexing or duplicating the sound hardware.

5.3 Larger Input Coverage

GF's *variants* feature allows an abstract syntax function to have multiple representations in a given concrete syntax. This permits some variation in the input, while producing the same interlingua term. For example, the linearization of PainKiller in the English concrete syntax in Figure 3 could be changed to:

lin PainKiller = *mkNP a_Art* (*mkN* "painkiller" | *mkN* "analgesic");

6 Conclusions

Because it uses a domain-specific interlingua, a GF-based speech translator can achieve high precision translation and scale to support a large number of languages.

The GF resource grammar library reduces the development effort needed to implement a speech translator for a new domain, and the need for the developer to have detailed linguistic knowledge.

Systems created with GF are highly portable to new platforms, because of the wide speech recognition grammar format support, and the availability of PGF interpreters for many platforms.

With additional work, GF could be used to implement a full-scale speech translator. The existing GF components for grammar development, speech recognition grammar compilation, parsing, and linearization could also be used as parts of larger systems.

References

Krasimir Angelov, Björn Bringert, and Aarne Ranta. PGF: A Portable Run-Time Format for Type-Theoretical Grammars. *Journal of Logic, Language and Information*, submitted, 2008.

P. Bouillon, M. Rayner, N. Chatzichrisafis, B. A. Hockey, M. Santaholma, M. Starlander, H. Isahara, K. Kanzaki, and Y. Nakao. A generic Multi-Lingual Open Source Platform for Limited-Domain Medical Speech Translation. In *Proceedings of the tenth Conference on European Association of Ma-*

chine Translation (EAMT 2005), Budapest, Hungary, pages 5–58, May 2005. URL http://www.issco.unige.ch/pub/MedSLT_demo_EAMT05_final.pdf.

Björn Bringert. Speech Recognition Grammar Compilation in Grammatical Framework. In *Proceedings of the Workshop on Grammar-Based Approaches to Spoken Language Processing, Prague, Czech Republic*, pages 1–8. Association for Computational Linguistics, June 2007. URL http://www.aclweb.org/anthology/W/W07/W07-1801.

Janna Khegai. Grammatical Framework (GF) for MT in sublanguage domains. In *Proceedings of EAMT-2006, 11th Annual conference of the European Association for Machine Translation, Oslo, Norway*, pages 95–104, June 2006. URL http://www.mt-archive.info/EAMT-2006-Khegai.pdf.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004. ISSN 0956-7968. doi: 10.1017/S0956796803004738.

Aarne Ranta. Grammars as software libraries. In Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin, editors, *From semantics to computer science: essays in honor of Gilles Kahn*. Cambridge University Press, 2008. URL http://www.cs.chalmers.se/~aarne/articles/libraries-kahn.pdf.

Paper V

Interactive Multilingual Web Applications with Grammatical Framework

GoTAL 2008, Gothenburg

Interactive Multilingual Web Applications with Grammatical Framework

Moisés Salvador Meza Moreno
Björn Bringert
Department of Computer Science and Engineering
Chalmers University of Technology
and University of Gothenburg
meza@student.chalmers.se, bringert@chalmers.se

Abstract

We present an approach to multilingual web content based on multilingual grammars and syntax editing for a controlled language. Content can be edited in any supported language and it is automatically kept within a controlled language fragment. We have implemented a web-based syntax editor for Grammatical Framework (GF) grammars which allows both direct abstract syntax tree manipulation and text input in any of the languages supported by the grammar. With this syntax editor and the GF JavaScript API, GF grammars can be used to build multilingual web applications. As a demonstration, we have implemented an example application in which users can add, edit and review restaurants in English, Spanish and Swedish.

1 Introduction

Current multilingual web applications store a separate version of their content for each language. It is difficult to keep the information consistent and, in some cases, content available in one language is not provided in another. Adding a new language to the application requires translation of the available content from one of the existing languages to the new language.

We suggest a different approach to multilingual web applications, where the content is defined by a multilingual grammar and is created through syntax editing or parsing. Content created by a user who uses one language is automatically available in all the other languages supported by the grammar, and the content is consistent at all times. When the grammar is extended to cover a new language, all existing content is automatically available in that language.

To demonstrate this approach to multilinguality, we have implemented "The Restaurant Review Wiki", a web-based multilingual application in which users

can add, edit and review restaurants in English, Spanish and Swedish. It uses GF grammars and the GF JavaScript API to provide multilinguality.

2 Grammatical Framework

Grammatical Framework (GF) (Ranta 2004) is a type-theoretical grammar formalism. GF grammars can describe both formal and natural languages and consist of an abstract syntax and at least one concrete syntax. The abstract syntax defines the scope of the grammar, i.e. all the expressions that can be built from it. The concrete syntax defines how the constructs in the abstract syntax are represented in a particular language. GF grammars can be multilingual, each language in the grammar having a separate concrete syntax. For any given grammar, GF provides parsing (going from a concrete to the abstract syntax) and linearization (going from the abstract to a concrete syntax). GF supports dependently typed and higher-order abstract syntax. These features are used, for example, to express conditions of semantic well-formedness. However, they are not used in this article since they are not supported in the implementations described.

GF includes a Resource Grammar Library (Ranta 2008) which defines the basic grammar of (currently) eleven languages. For each language, the Resource Grammar Library provides the complete morphology, a lexicon of approximately one hundred of the most important structural words, a test lexicon of approximately 300 content words, a list of irregular verbs and a substantial fragment of the syntax. The Resource Grammar Library has an API (Application Programming Interface) which allows the user to implement grammars for these languages easily. The API also provides tools to extend the resource grammars, for example, new words can be added to the lexicon. GF is freely available[1] and is distributed under the GNU General Public License (GPL).

2.1 An Example Grammar

To better explain GF grammars, consider a very small grammar that describes simple restaurant reviews. The abstract syntax defines what can be said in the grammar in terms of categories (**cat**) and functions (**fun**). In the example grammar, the abstract syntax (Figure 1) has four categories: Phrase (the start category), Item, Demonym and Quality. It also has some functions that construct terms in these categories. For example, the function *itemIs* takes an Item and a Quality as arguments and produces a Phrase, and an Item can be either *restaurant* or *food*. Examples of abstract terms produced by this abstract syntax are *itemIs* (*qualItem mexican food*) (*very good*) and *itemIs restaurant expensive*.

The concrete syntax specifies how the different abstract syntax terms are expressed in a particular language. There is a linearization type (**lincat**) for every category in the abstract syntax. The linearization type is the type of the concrete syntax terms produced for the abstract syntax terms in a category.

[1] `http://digitalgrammars.com/gf/`

```
abstract Restaurant = {
  flags startcat = Phrase;
  cat Phrase; Item; Demonym; Quality;
  fun itemIs              : Item → Quality → Phrase;
      restaurant, food    : Item;
      qualItem            : Demonym → Item → Item;
      italian, mexican    : Demonym;
      very                : Quality → Quality;
      good, bad, cheap, expensive : Quality;
}
```

Figure 1. Abstract syntax for the example grammar.

Similarly, there is a linearization definition (**lin**) for every function in the abstract syntax. A linearization definition is a function from the linearizations of the arguments of an abstract syntax function to a concrete syntax term.

Figure 2 shows the English concrete syntax for the example grammar. The linearization type for all categories is $\{s : \mathsf{Str}\}$, that is, a record with a single field s of type Str (string). The linearization of the function *restaurant* is the concrete syntax term $\{s =$ "restaurant"$\}$. The linearization of *itemIs* makes use of the linearizations of its argument terms of type Item and Quality. The linearization of the abstract syntax term *itemIs restaurant expensive* is the string "the restaurant is expensive".

Figure 3 shows the Spanish concrete syntax for the example grammar. This concrete syntax is more complex because Spanish nouns have an inherent gender (masculine or feminine). Adjectives are inflected according to the gender of the noun they modify and the form of the definite article depends on the gender of the noun it modifies. Thus the category Item has a linearization type $\{s : \mathsf{Str}; g : \mathsf{Gender}\}$. In addition to the string field s, the record has a field g of type Gender, either Masc or Fem. The categories Demonym and Quality have a linearization type $\{s : \mathsf{Gender} \Rightarrow \mathsf{Str}\}$. The field s is here a function from Gender to Str. Some helper functions (**oper**) are also defined. For example, the function *adjective* takes a Str and returns a record of type $\{s : \mathsf{Gender} \Rightarrow \mathsf{Str}\}$. The abstract syntax term *itemIs* (*qualItem mexican food*) (*very good*) is linearized to "la comida mexicana es muy buena". If we replace the feminine noun *food* with the masculine noun *restaurant* the linearization changes to "el restaurante mexicano es muy bueno".

To write the Spanish concrete syntax, the grammar writer had to take into account the morphological and syntactic features of the Spanish language. Even in this simple example, gender had to be considered; imagine a grammar in which number plus case is also involved, or polarity, or verb conjugation, or all of them at once. The larger the scope of the grammar, the harder it gets to properly handle the features of a language. That is why GF's Resource

```
concrete RestaurantEng of Restaurant = {
  lincat Phrase, Item, Demonym, Quality = {s : Str};
  lin itemIs i q    = {s = "the" ++ i.s ++ "is" ++ q.s};
      restaurant    = {s = "restaurant"};
      food          = {s = "food"};
      qualItem d i  = {s = d.s ++ i.s};
      italian       = {s = "Italian"};
      mexican       = {s = "Mexican"};
      very q        = {s = "very" ++ q.s};
      good          = {s = "good"};
      bad           = {s = "bad"};
      cheap         = {s = "cheap"};
      expensive     = {s = "expensive"};
}
```

Figure 2. English concrete syntax for the example grammar.

Grammar Library was implemented: to define the low-level morphological and syntactic rules of languages and allow grammar writers to focus on the domain-specific semantic and stylistic aspects. The idea is that if a grammar uses the Resource Grammar Library in a type correct way, it will produce grammatically correct output. The grammar writer still has to know the target language and the application domain in order to get the semantics and pragmatics right, since the grammar library only handles syntax and morphology. Figure 4 shows a Spanish concrete syntax for the example grammar which uses the Resource Grammar Library. The categories Phrase, Item, Demonym and Quality have the linearization types Phr (phrase), CN (common noun), A (one-place adjective) and AP (adjectival phrase), respectively. All linearizations use functions from the resource grammar, such as $mkN : \mathsf{Str} \to \mathsf{N}$, $mkA : \mathsf{Str} \to \mathsf{A}$ and $mkNP : \mathsf{Det} \to \mathsf{N} \to \mathsf{NP}$.

3 Syntax Editing

A *syntax editor* (also known as *syntax-directed editor*, *language-based editor*, or *structure editor*) lets the user edit documents by manipulating their underlying structure. Such editors can be constructed for any type of structured document, for example computer programs (Teitelbaum and Reps 1981), or structured text documents (Furuta et al. 1988).

In the context of GF, a syntax editor lets the user manipulate abstract syntax terms for a particular grammar, while displaying its linearization(s). Syntax editing with GF grammars is described in more detail by Khegai et al. (2003). To explain GF syntax editing we will make use of the grammar described in Section 2.1. There are two kinds of abstract syntax terms: complete terms,

```
concrete RestaurantSpa of Restaurant = {
  lincat Phrase              = {s : Str};
         Item                = {s : Str; g : Gender};
         Demonym, Quality    = {s : Gender ⇒ Str};
  lin itemIs i q     = {s = defArt ! i.g ++ i.s ++ "es" ++ q.s ! i.g};
      restaurant     = {s = "restaurante"; g = Masc};
      food           = {s = "comida"; g = Fem};
      qualItem d i   = {s = i.s ++ d.s ! i.g; g = i.g};
      italian        = adjective "italiano";
      mexican        = adjective "mexicano";
      very qual      = {s = \\g ⇒ "muy" ++ qual.s ! g};
      good           = adjective "bueno";
      bad            = adjective "malo";
      cheap          = adjective "barato";
      expensive      = adjective "caro";
  param Gender = Masc | Fem;
  oper defArt : Gender ⇒ Str = table {Masc ⇒ "el"; Fem ⇒ "la"};
       adjective : Str → {s : Gender ⇒ Str} =
         λx → {s = table {Masc ⇒ x; Fem ⇒ Predef.tk 1 x + "a"}};
}
```

Figure 3. Spanish concrete syntax for the example grammar.

e.g. *itemIs restaurant good* and incomplete terms, e.g. *itemIs food* ?. A question mark in an incomplete term is a *metavariable*, i.e. a non-instantiated term. The metavariable in the incomplete term *itemIs food* ? is of type Quality. Syntax editing starts with a single metavariable and it is refined step-by-step until the desired complete term is constructed.

4 GF JavaScript Syntax Editor

This is a syntax editor written in JavaScript that can be used in any JavaScript enabled web browser. This allows the syntax editor to be embedded into web applications. It can also be used as a complete application by itself, for example, to explore, debug or test GF grammars interactively.

4.1 User Interface

The editor interface contains six panels (Figure 5):

Abstract syntax tree panel Shows a tree representation of the abstract syntax term being edited. Selecting a node will highlight both the node in this panel and its corresponding linearization(s) in the linearization panel.

```
concrete RestaurantSpaRes of Restaurant = open TrySpa in {
  lincat Phrase = Phr; Item = CN; Demonym = A; Quality = AP;
  lin itemIs i q    = mkPhr (mkCl (mkNP the_Art i) q);
      restaurant    = mkCN (mkN "restaurante");
      food          = mkCN (mkN "comida");
      qualItem d i  = mkCN d i;
      italian       = mkA "italiano";
      mexican       = mkA "mexicano";
      very qual     = mkAP very_AdA qual;
      good          = mkAP (mkA "bueno");
      bad           = mkAP (mkA "malo");
      cheap         = mkAP (mkA "barato");
      expensive     = mkAP (mkA "caro");
}
```

Figure 4. Spanish concrete syntax using the resource grammar library.

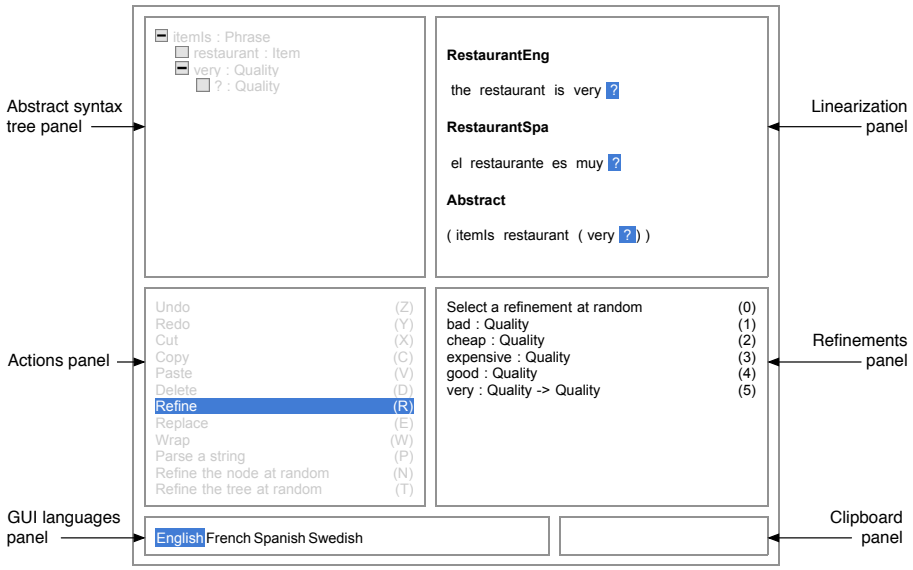

Figure 5. GF JavaScript syntax editor.

Linearization panel Shows the linearizations of the current abstract syntax term in all the available concrete syntaxes. A string representation of the abstract syntax term is also shown. Clicking on a word in a linearization will select the corresponding node in the tree shown in the abstract syntax tree panel. Metavariables are linearized as question marks.

Actions panel Used to show the actions available for the selected node (see Section 4.2). Actions not available for the selected node are grayed out.

Refinements panel Used to show the available refinements or wrappers for the selected node whenever the "Refine" or "Wrap" action is selected.

GUI languages panel Used to show and select the different languages available for the GUI (Graphical User Interface). Currently, three languages are supported: English, Spanish and Swedish. The goal is to support all the languages in GF's Resource Grammar Library. This interface localization is implemented using the approach described in Section 5.2.

Clipboard panel Used to show the name and type of the term currently stored in the clipboard. The clipboard only holds one term at any given time.

4.2 Syntax Editing Actions

There are a number of actions that can be performed on abstract syntax terms. Some of the actions require no further explanation, among those we find: *Undo*, *Redo*, *Cut*, *Copy* and *Paste*. Some of the actions can be easily explained: *Delete* replaces an instantiated term with a metavariable, *Replace* is equivalent to *Delete* followed by *Refine*, except that it is treated as a single action in the edit history and *Refine the node at random* and *Refine the tree at random* respectively instantiate every metavariable in the subtree rooted at the selected node and the entire abstract syntax tree with type-correct objects selected at random. Finally, the following actions deserve a more in depth description:

Refine Replaces a metavariable with a function of the appropriate type. The arguments of the function will all be metavariables. To refine a metavariable of type Phrase (Figure 6(a)) we need to choose one function from those that have the return type Phrase. Only the function *itemIs* : Item → Quality → Phrase fits this requirement. This refinement will yield a term of the form *itemIs* ? ? where the metavariables are of type Item and Quality (Figure 6(b)).

Wrap Replaces an instantiated term of type T with a function which has at least one argument of type T and a return type T. The original term is used as the child corresponding to the first argument of type T; the remaining children will be metavariables. In the example grammar, any term of type Quality can be wrapped with the function *very* : Quality → Quality. Wrapping the term *good* of type Quality, shown in Figure 7(a), with the function *very* (Figure 7(b)) results in the term *very good* of type Quality

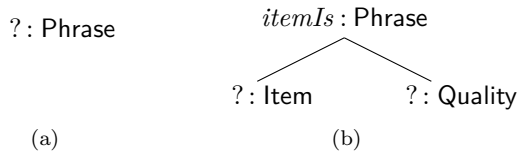

Figure 6. Refining a metavariable of type Phrase.

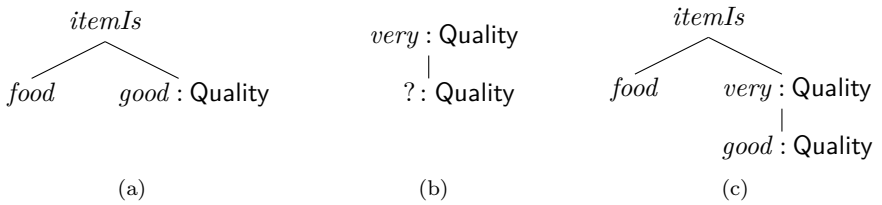

Figure 7. Wrapping the abstract term *good*.

(Figure 7(c)). There is one exception: the top level node can be wrapped by any function which has at least one argument of type T regardless of its return type.

Parse a string Prompts the user for a string and tries to generate a type-correct subtree by parsing it. On success, the node is instantiated with the resulting subtree. GF grammars can be ambiguous, i.e. two abstract terms can have the same linearization. When parsing an ambiguous string, GF returns a list of abstract terms. In the syntax editor, the different trees produced when parsing an ambiguous string are displayed in the refinements panel so that the user can select one.

4.3 Implementation

We have implemented a GF JavaScript API that allows parsing, linearization, type-annotation of meta-variables, and abstract syntax tree serialization and deserialization to be done in JavaScript applications. This code is based on the existing GF JavaScript linearization implementation, which was originally used for output generation in GF-generated VoiceXML applications (Bringert 2007). We have extended it with parsing functionality, by using the active MCFG parsing algorithm described by Burden and Ljunglöf (Burden and Ljunglöf 2005).

The GF JavaScript API is now essentially an interpreter for PGF (Portable Grammar Format) (Angelov et al. 2008). PGF is a low-level format for type-theoretical grammars, and the main target of the GF grammar compiler. The GF grammar compiler has been extended to translate the PGF grammars it produces into a JavaScript representation, which is used by the GF JavaScript API. The JavaScript representation, which is isomorphic to the subset of PGF

needed for type-checking, parsing and linearization, is used instead of the standard PGF form in order to avoid the extra computation needed to read PGF files directly in JavaScript.

On top of this API, the syntax editor implements the syntax editing actions, and facilities for supporting the editor user interface. One interesting addition is the support for associating parts of the linearization output with the abstract syntax sub-terms which generated them. Each node in the abstract syntax tree is given an identifier which encodes the path from the root of the tree to the given node. The linearization algorithm has been modified to tag each token that results from linearizing a node with that node's identifier. As a consequence, each token in the sequence of tokens produced by linearizing an abstract syntax tree will be tagged with the identifier of the node that produced it, and the identifiers of all its parent nodes. When the user selects a node in the tree, all tokens tagged with that node's identifier are highlighted. When a token is selected, the deepest node (i.e. longest identifier) which it is tagged with is highlighted.

5 Example Application: The Restaurant Review Wiki

The GF JavaScript API and the syntax editor described in Section 4 can be used together to build a multilingual web application. This section describes the Restaurant Review Wiki, a small demo application developed using these tools.

5.1 Description

The Restaurant Review Wiki is a restaurant database that allows users to add restaurants and reviews and view and edit the information in three languages (English, Swedish and Spanish). It is available online[2].

Users can add new restaurants and edit the information about existing restaurants. For each restaurant there is some basic information, such as address and cuisine, entered using standard HTML forms, and reviews which are created and edited by using the syntax editor as shown in Figure 8. The restaurant review grammar used in this application is an extended version of the grammar described in Section 2.1.

When adding a new review, the abstract syntax term in the syntax editor is initially a single metavariable of type Paragraph. The user edits a review by stepwise refining the tree, by parsing a string or by some combination of these. For example, the user may parse a simple sentence such as "the food is delicious", and then use syntax editing commands to elaborate parts of it.

[2] `http://csmisc14.cs.chalmers.se/~meza/restWiki/wiki.cgi/`

Figure 8. Review editing page.

5.2 Implementation

Instead of storing the text in any language, the abstract syntax representation of the information is stored on the server and it is linearized by the client's browser upon request. The algorithms to linearize abstract syntax trees are efficient and with today's computing power the user should not be affected by delays caused by the linearization of the different multilingual elements of a page. Whenever a page is loaded, a linearizing function is called for every multilingual element in the page. This function takes the HTML element to linearize, a reference to the currently selected language and a grammar as arguments. It extracts the string representation of the abstract syntax term from the element, converts it into an abstract syntax tree, linearizes the tree using the concrete syntax for the currently selected language and stores the linearization in the element.

Two GF grammars are used by this application, one that describes the elements of web pages such as headers, field names, country names, cuisines, etc., and another that describes restaurant reviews.

5.3 Discussion

Advantages

Since the multilingual information is stored as its abstract syntax representation, all new content created by users is available for all languages immediately, and it is thereby consistent in all languages. In existing multilingual applications

such as Wikipedia, multilingual content is created in parallel. This means that there is a different version of the information for each language and there is no guarantee that the information available for a particular language will be available in another nor that they will be consistent.

Having all the information in an abstract representation of a controlled language makes it possible to perform operations such as querying precisely and efficiently. For example, it should be easy to implement functionality that would let the user search for "cheap Thai restaurants close to the university".

Adding a language to the application means adding a concrete syntax for that language to the grammar. Once the concrete syntax is added, all existing information is automatically available in the new language. There is no need to translate the existing information by hand.

Disadvantages

The content that can be created using this approach is limited by the coverage of the grammar. This may be too restrictive and it may prevent users from effectively conveying their ideas through the content they create.

In this version of the application, new content is created by using the syntax editor, either by stepwise refining the abstract syntax tree or by parsing a string. The syntax editor has the advantage of generating content within the coverage of the grammar. The problem is that the editor is not very intuitive and it could be hard to use without training, a situation that could discourage potential users. Creating content by parsing is simple, but, if the user is not familiar with the grammar, producing valid content through parsing might be a difficult task unless the grammar has a very wide coverage.

Multilingual processing is done in the client rather than on the server. A JavaScript GF grammar may be larger than 1 MB, which could be a problem for devices with limited bandwidth or memory, such as PDAs or mobile phones. Also, devices with limited processing power may experience delays caused by the linearization of the multilingual elements in pages. Since the current version does linearization in the client even when viewing existing content, search engines may not be able to index the page using the linearized content.

If an abstract syntax used in an application is changed and the new version is not backwards compatible, it may no longer be possible to linearize the stored abstract syntax terms. If the coverage of the new grammar is a superlanguage of the old one, this problem can be solved by linearizing each stored term with the old grammar and parsing it with the new one.

Doing natural language processing client-side tends to stress the web browser implementations. The current state of web standards compatibility in browsers may lead to inconsistent behavior or performance in some web browsers.

6 Related Work

The Grammatical Framework (GF) provided, up until this point, two different syntax editors. The first provides the full functionality of GF but can only be used in machines that have the full GF system installed (Ranta 2004). One use of this editor is as an integral component of the KeY formal program verification system (Beckert et al. 2007). The second, Gramlets (Johannisson et al. 2003), provides no parsing and no support for dependent types or higher-order functions but can be run on any machine that has a Java Virtual Machine (JVM) installed or in web browsers which have a JVM plug-in. Our syntax editor is more portable than the previous GF syntax editors, can be more easily integrated into web applications, and compared to Gramlets, it offers more functionality, most notably parsing. The syntax editor does not support the full GF language yet, as it only allows grammars which have no dependent types and no higher-order abstract syntax.

WYSIWYM (Power et al. 1998) is a structure editor which displays natural language representations during editing. It now also has a JavaScript implementation[3]. Our editor is driven by a declarative specification of the language structure and generation rules. In WYSIWYM these components are built into the editor, which appears to make it more difficult to use the editor for new applications.

7 Future Work

Dependently Typed and Higher-order Abstract Syntax For the syntax editor to support more advanced grammars, the GF JavaScript API should be extended to implement parsing, type-checking and linearization for grammars with dependently typed and higher-order abstract syntax.

Syntax Editor User Interface New content is created using the syntax editor and, as mentioned before, this is too restrictive and could make users lose interest in the application. There is a need for a more intuitive interface which still guarantees that the content is within the domain of the grammar. One way to make the interface more easy to use is to add *completion*. The idea is to make the editor display a list of possible ways to complete the input that the user is typing, as is done in the GF-based WebALT exercise editor for multilingual mathematical exercises (Cohen et al. 2006).

Server-side Processing Instead of doing the multilingual processing in the client, it could be done on the server. This would be beneficial for devices with limited processing power, memory or bandwidth. Especially linearization of existing content should be off-loaded to the server, as this will also help search engines index the content.

[3] `http://www.itri.brighton.ac.uk/projects/WYSIWYM/javademo.html`

8 Conclusions

We have implemented a syntax editor which provides the basic functionality of the Grammatical Framework (GF) in web browsers. It allows the user to stepwise create the abstract syntax trees described by a GF grammar through the use of special purpose editing actions, while showing linearizations of the trees in multiple languages. It can be used to test and debug GF grammars, or as a component in multilingual web-based applications.

To demonstrate how the syntax editor can be used to implement multilingual web applications, we also implemented "The Restaurant Review Wiki". It is a multilingual restaurant database in which users can add, edit and review restaurants in three different languages. The approach to multilinguality that we suggest makes all information available simultaneously and consistently for all the supported languages, and adding a new language is only a matter of adding a concrete syntax for that language to the application grammar. Additional work is required to make syntax editing more usable for untrained users, and to ensure that the technique works well in resource-constrained computing devices.

References

Krasimir Angelov, Björn Bringert, and Aarne Ranta. PGF: A Portable Run-Time Format for Type-Theoretical Grammars. *Journal of Logic, Language and Information*, submitted, 2008.

Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, Heidelberg, April 2007. doi: 10.1007/978-3-540-69061-0.

Björn Bringert. Rapid Development of Dialogue Systems by Grammar Compilation. In Simon Keizer, Harry Bunt, and Tim Paek, editors, *Proceedings of the 8th SIGdial Workshop on Discourse and Dialogue, Antwerp, Belgium*, pages 223–226. Association for Computational Linguistics, September 2007. URL `http://www.sigdial.org/workshops/workshop8/Proceedings/SIGdial39.pdf`.

Håkan Burden and Peter Ljunglöf. Parsing Linear Context-Free Rewriting Systems. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 11–17, Vancouver, British Columbia, 2005. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W05/W05-1502`.

Arjeh Cohen, Hans Cuypers, Karin Poels, Mark Spanbroek, and Rikko Verrijzer. WExEd - WebALT Exercise Editor for Multilingual Mathematical Exercises. In Mika Seppälä, Sebastian Xambo, and Olga Caprotti, editors, *WebALT 2006, First WebALT Conference and Exhibition, Eindhoven, The*

Netherlands, pages 141–145, January 2006. URL `http://www.win.tue.nl/~amc/pub/wexed.pdf`.

R. Furuta, V. Quint, and J. Andre. Interactively Editing Structured Documents. *Electronic Publishing*, 1(1):19–44, 1988. URL `http://cajun.cs.nott.ac.uk/wiley/journals/epobetan/pdf/volume1/issue1/eprxf011.pdf`.

Kristofer Johannisson, Janna Khegai, Markus Forsberg, and Aarne Ranta. From Grammars to Gramlets. In *The Joint Winter Meeting of Computing Science and Computer Engineering*. Chalmers University of Technology, 2003.

Janna Khegai, Bengt Nordström, and Aarne Ranta. Multilingual Syntax Editing in GF. In Alexander Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing*, volume 2588 of *Lecture Notes in Computer Science*, pages 199–204. 2003. doi: 10.1007/3-540-36456-0_48.

Richard Power, Donia Scott, and Roger Evans. What You See Is What You Meant: direct knowledge editings with natural language feedback. In Henri Prade, editor, *13th European Conference on Artificial Intelligence (ECAI 1998)*, pages 677–681, Chichester, England, 1998. John Wiley and Sons. URL `http://citeseer.ist.psu.edu/power98what.html`.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004. ISSN 0956-7968. doi: 10.1017/S0956796803004738.

Aarne Ranta. Grammars as software libraries. In Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin, editors, *From semantics to computer science: essays in honor of Gilles Kahn*. Cambridge University Press, 2008. URL `http://www.cs.chalmers.se/~aarne/articles/libraries-kahn.pdf`.

Tim Teitelbaum and Thomas Reps. The Cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, September 1981. ISSN 0001-0782. doi: 10.1145/358746.358755.

Paper VI

PGF: A Portable Run-Time Format for Type-Theoretical Grammars

Submitted to the Journal of Logic, Language and Information

PGF: A Portable Run-Time Format for Type-Theoretical Grammars

Krasimir Angelov Björn Bringert
Aarne Ranta
Department of Computer Science and Engineering
Chalmers University of Technology
and University of Gothenburg
SE-412 96 Göteborg, Sweden
{krasimir,bringert,aarne}@chalmers.se

Abstract

PGF (Portable Grammar Format) is a low-level language used as a target of compiling grammars written in GF (Grammatical Framework). Low-level and simple, PGF is easy to reason about, so that its language-theoretic properties can be established. It is also easy to write interpreters that perform parsing and generation with PGF grammars, and compilers converting PGF to other format. This paper gives a concise description of PGF, covering syntax, semantics, and parser generation. It also discusses the technique of embedded grammars, where language processing tasks defined by PGF grammars are integrated in larger systems.

1 Introduction

PGF (Portable Grammar Format) is a grammar formalism designed to capture the computational core of type-theoretical grammars, such as those written in GF (Grammatical Framework, Ranta 2004). PGF thus relates to GF in the same way as JVM (Java Virtual Machine) bytecode relates to Java. While GF (like Java) is a rich language, whose features help the programmer to express her ideas on a high level of abstraction, PGF (like JVM) is an austere language, which is easy to implement on a computer and easy to reason about. The bridge between these two level, in both cases, is a compiler. The compiler gives the grammar writer the best of the two worlds: when writing grammars, she can concentrate on linguistic ideas and find concise expressions for them; when using grammars, she can enjoy efficient run-time performance and a light-weight run-time system, as well as integration in applications.

PGF was originally designed as a back-end language for GF, providing the minimum of what is needed to perform generation and parsing. Its expressive

power is between context-free and fully context-sensitive (Ljunglöf 2004). Thus it can potentially provide a back end to numerous other grammar formalisms as well, providing for free a large part of what is involved in implementing these formalisms. In addition, PGF can be compiled further into other formats, such as language models for speech recognition systems (Bringert 2007a).

The most prominent characteristic of PGF (as well as GF) is **multilinguality**: a PGF grammar has an **abstract syntax**, which is a type-theoretical definition of tree structures. The abstract syntax is equipped with a set of **concrete syntaxes**, which are reversible mappings of tree structures to strings. This grammar architecture is known as the Curry architecture, with a reference to Curry (1961). It is at the same time familiar from programming language descriptions, dating back to McCarthy (1962). While the Curry architecture was used by Montague (1974) for describing English, GF might be the first grammar formalism that exploits the multilingual potential of the Curry architecture.

Multilingual PGF grammars readily support translation and multilingual generation. They are also useful when writing monolingual applications, since the non-grammar parts of an application typically communicate with the abstract syntax alone. This follows the standard architecture of compilers, which use an abstract syntax as the format in which a programming language is presented to the other components, such as the type checker and the code generator (Appel 1997). Thus an application using PGF is easy to port from one target language to another, by just writing a new concrete syntax. The GF **resource grammar library** (Ranta 2008) helps application programmers by providing a comprehensive implementation of syntactic and morphological rules for 15 languages.

JVM is a general-purpose Turing-complete language, but PGF is limited to expressing grammars. Grammars can be seen as declarative programs performing tasks such as parsing and generation. In a language processing system, these tasks usually have to be combined with other functionalities. For instance, a question answering system reads input and produces output by using grammars, but the program that computes the answers from the questions has to be written in another language. **Embedded grammars** is a technique that enables programs written in another programming language to call PGF functionalities and also to manipulate PGF syntax trees as data objects. There are two main ways to implement this idea: interpreters and compilers. A PGF interpreter is a program written in a general-purpose language (such as C++, Haskell, or Java, which we have already written interpreters for). The interpreter reads a PGF grammar from a file and gives access to parsing and generation with the grammar. A PGF compiler translates PGF losslessly to another language (such as C, JavaScript, and Prolog, for which compilers have already been written). When we say that PGF is portable, we mean that one can write PGF interpreters and compilers for different host languages, so that the same PGF grammars can be used as components in applications written in these host languages.

The purpose of this paper is to describe the PGF grammar format and briefly show how it is used in applications. The PGF description is detailed enough to enable the reader to write her own PGF interpreters and compilers; but it is

informal in the sense that we don't fully specify the concrete format produced by the current GF-to-PGF compiler and implemented by the current PGF interpreters. For that level of detail, on-line documentation is more appropriate and can be found on the GF web page[1].

Section 2 gives a concise but complete description of the syntax and semantics of PGF. Section 3 gives a summary of the expressive power of PGF, together with examples illustrating its main properties. It also discusses some extensions of PGF. Section 4 describes the parser generation and sketches the parsing algorithm of PGF. Section 5 discusses the principal ways of using PGF in language processing applications, the compilation from PGF to other formats, and the compilation of PGF grammars from GF, which is so far the main way to produce PGF grammars. Section 6 gives a survey of actual applications running PGF and provides some data evaluating its performance. Section 7 discusses related work, and Section 8 gives a conclusion.

2 The syntax and semantics of PGF

2.1 Multilingual grammar

The top-most program unit of PGF is a **multilingual grammar** (briefly, grammar). A grammar is a pair of an **abstract syntax** $\mathcal{A}$ and a set of **concrete syntaxes** $\mathcal{C}_1, \ldots, \mathcal{C}_n$, as follows:

$$\mathcal{G} = < \mathcal{A}, \{\mathcal{C}_1, \ldots, \mathcal{C}_n\} >$$

2.2 Abstract syntax

An abstract syntax has a finite a set of **categories** and a finite set of **functions**. Categories are defined simply as unique identifiers, whereas functions are unique identifiers equipped with **types**. A type has the form

$$(\mathsf{C}_1, \ldots, \mathsf{C}_n) \to \mathsf{C}$$

where n may be 0 and each of $\mathsf{C}_1, \ldots, \mathsf{C}_n, \mathsf{C}$ is a category. These types are actually **function types** with **argument types** $\mathsf{C}_1, \ldots, \mathsf{C}_n$ and **value type** C.

Here is an example of an abstract syntax, where we use the keyword **cat** to give the categories and **fun** to give the functions. It defines the categories S (sentence), NP (noun phrase), and VP (verb phrase). Sentences are formed by the Pred function. John is given as an example of a noun phrase and Walk of a verb phrase.

cat S; NP; VP
fun Pred : (NP, VP) → S
fun John : () → NP
fun Walk : () → VP

[1] `http://gf.digitalgrammars.com/`

An **abstract syntax tree** (briefly, tree) is formed by applying the functions obeying their typings, as in simply typed lambda calculus. Thus, for instance, $\mathsf{Pred}\ (\mathsf{John}, \mathsf{Walk})$, is a tree of type S, usable for representing the string *John walks*.

2.3 Concrete syntax

A concrete syntax has judgements assigning a **linearization type** to each category in the abstract syntax and a **linearization function** to each function. Linearization types are **tuples** built from **strings** and **bounded integers**. Thus we have the following forms of linearization types T:

$$\mathsf{T} ::= \mathsf{Str} \mid \mathsf{Int}_n \mid \mathsf{T}_1 * \ldots * \mathsf{T}_n$$

Linearization functions are **terms** t of these types, built in the following ways:

$$t ::= [\,] \mid \text{“foo”} \mid t_1 ++ t_2 \mid i \mid <t_1, \ldots, t_n> \mid t_1 \,!\, t_2 \mid \$_i$$

The first three forms are canonical for strings: the empty string $[\,]$, a token (quoted, like “foo”), and the **concatenation** $++$. Concatenation is associative, and it preserves the separation between tokens. The empty string is nilpotent with respect to concatenation. The canonical PGF representation of the string *John loves Mary*, if conceived as consisting of three tokens, is thus “John” $++$ “loves” $++$ “Mary”.

The form i, where i is a numeric constant $1, 2, \ldots, n$, is canonical for integers bounded by n. The form $<t_1, \ldots, t_n>$ is canonical for tuples, comprising a term t_i for each component type T_i in a tuple type.

The last two forms in the term syntax are non-canonical. The form $t_1 \,!\, t_2$ is the **projection** of t_2 from t_1. In order for the projection to be well-formed, t_1 must be a tuple and t_2 an integer within the bounds of the size of the tuple. The last form $\$_i$ is an **argument variable**. It is bound to a term given in a context, which, as we shall see, always consists of the linearizations of the subtrees of a tree.

Table 1 gives the static typing rules of the terms in PGF. It defines the relation $\Gamma \vdash t : T$ (“in context Γ, term t has type T”). The context is a sequence of types, and it is left implicit in all rules except the one for argument variables.

The context is in each linearization function created from the linearization types of the arguments of the function. Thus, if we have

fun $f : (\mathsf{C}_1, \ldots, \mathsf{C}_n) \to \mathsf{C}$
lincat $\mathsf{C}_1 = \mathsf{T}_1; \ldots; \mathsf{C}_n = \mathsf{T}_n$
lincat $\mathsf{C} = \mathsf{T}$
lin $f = t$

then we must also have

$$T_1, \ldots, T_n \vdash t : T$$

Strings:

$$[\,] : \mathsf{Str} \quad \text{“foo”} : \mathsf{Str} \quad \frac{s, t : \mathsf{Str}}{s \mathbin{+\!\!+} t : \mathsf{Str}}$$

Bounded integers:

$$i : \mathsf{Int}_i \quad \frac{i : \mathsf{Int}_m}{i : \mathsf{Int}_n}\ m < n$$

Tuples:

$$\frac{t_1\ :\ T_1\ \ldots\ t_n\ :\ T_n}{< t_1, \ldots, t_n > :\ T_1 * \cdots * T_n}$$

Projections:

$$\frac{t : \mathsf{T}^n \quad u : \mathsf{Int}_n}{t\,!\,u : \mathsf{T}} \qquad \frac{t : \mathsf{T}_1 * \ldots * \mathsf{T}_n}{t\,!\,i : \mathsf{T}_i}\ i = 1, \ldots, n$$

Argument variables:

$$\frac{}{T_1, \ldots, T_n \ \vdash\ \$_i : \mathsf{T}_i}\ i = 1, \ldots, n$$

Table 1. The type system of PGF concrete syntax.

The alert reader may notice that the typing rules for projections are partial: they cover only the special cases where either all types in the tuple are the same (denoted T^n) or the index is an integer constant. If the types are different and the index has an unknown value (which happens when it e.g. depends on an argument variable), the type of the projection cannot be known at compile time. As we will see in Section 5.4, PGF grammars compiled from GF always fall under these special cases.

2.4 Examples of a concrete syntax

Let us build a concrete syntax for the abstract syntax of the Section 2.2. We define the linearization type of sentences to be just strings, whereas noun phrases and verb phrases are more complex, to account for **agreement**. Thus a noun phrase is a pair of a string and an agreement feature, where the feature is represented by an integer. A verb phrase is a tuple of strings—as many as there are possible agreement features. In this simple example, we just assume two features, corresponding to the singular and the plural.

lincat $\mathsf{S} = \mathsf{Str}; \mathsf{NP} = \mathsf{Str} * \mathsf{Int}_2; \mathsf{VP} = \mathsf{Str} * \mathsf{Str};$

The linearization of John is the string “John” with the feature 1, representing the singular. The linearization of Walk gives the two forms of the verb *walk*.

lin $\mathsf{John} = <\text{“John”}, 1>; \mathsf{Walk} = <\text{“walks”}, \text{“walk”} >$

The agreement itself is expressed as follows: in predication, the first field (1) of the noun phrase ($\$_1$)—is concatenated with a field of the verb phrase ($\$_2$). The field that is selected is given by the second field of the first argument ($\$_1\,!\,2$):

Strings:

$$[\,] \Downarrow [\,] \quad \text{“foo”} \Downarrow \text{“foo”} \quad \frac{s \Downarrow v \quad t \Downarrow w}{s \mathbin{+\!\!+} t \Downarrow v \mathbin{+\!\!+} w}$$

Bounded integers:

$$i \Downarrow i$$

Tuples:

$$\frac{t_1 \Downarrow v_1 \ \ldots \ t_n \Downarrow v_n}{< t_1, \ldots, t_n > \Downarrow < v_1, \ldots, v_n >}$$

Projections:

$$\frac{t \Downarrow < v_1, \ldots, v_n > \quad u \Downarrow i}{t!u \Downarrow v_i} \; i = 1, \ldots, n$$

Argument variable:

$$v_1, \ldots, v_n \vdash \$_i \Downarrow v_i \; (i = 1, \ldots, n)$$

Table 2. The operational semantics of PGF.

$$\textbf{lin}\ \mathsf{Pred} = \$_1\,!\,1 \mathbin{+\!\!+} \$_2\,!\,(\$_1\,!\,2)$$

The power of a multilingual grammar comes from the fact that both linearization types and linearization functions are defined independently in each concrete syntax. Thus German, in which both a two-value number and a three-value person are needed in predication can be given the following concrete syntax:

$$\begin{array}{l}
\textbf{lincat}\ \mathsf{NP} = \mathsf{Str} * \mathsf{Int}_2 * \mathsf{Int}_3 \\
\qquad\ \mathsf{VP} = (\mathsf{Str} * \mathsf{Str} * \mathsf{Str}) * (\mathsf{Str} * \mathsf{Str} * \mathsf{Str}) \\
\textbf{lin}\ \mathsf{Pred} = \$_1\,!\,1 \mathbin{+\!\!+} (\$_2\,!\,(\$_1\,!\,2))\,!\,(\$_1\,!\,3) \\
\quad \mathsf{John} = <\text{“John”}, 1, 3> \\
\quad \mathsf{Walk} = << \text{“gehe”}, \text{“gehst”}, \text{“geht”}>, <\text{“gehen”}, \text{“geht”}, \text{“gehen”} >>
\end{array}$$

2.5 Linearization

Linearization—the operation $\mapsto$ converting trees into concrete syntax objects—is performed by following the operational semantics given in Table 2. The table defines the relation $\gamma \vdash t \Downarrow v$ (“in context γ, term t evaluates to term v”). The context is a sequence of terms, and it is left implicit in all rules except the one for argument variables.

To linearize a tree, we linearize its immediate subtrees, and the sequence of the resulting terms constitutes the context of evaluation for the full tree:

$$\frac{a_1 \mapsto t_1 \ \ldots \ a_n \mapsto t_n \quad t_1, \ldots, t_n \vdash t \Downarrow v}{f\, a_1 \ldots a_n \mapsto v}\, \textbf{lin}\, f = t$$

Since linearization operates on the linearizations of immediate subtrees, it is a homomorphism, in other words, a compositional mapping. A crucial property

of PGF making it possible to maintain the compositionality of linearization in realistic grammars is that its value need not be a string, but a richer data structure. If strings are what ultimately are needed, one can require that the start category has the linearization type Str; alternatively, one can define a realization operation that finds the first string in a tuple recursively.

3 Properties of PGF

3.1 Expressive power

Context-free grammars have a straightforward representation in PGF: consider a rule

$$C \longrightarrow t_1 \ldots t_n$$

where every t_i is either a nonterminal C_j or a terminal s. This rule can be translated to a pair of an abstract function and a linearization:

fun $f : (\mathsf{C}_1, \ldots, \mathsf{C}_m) \to \mathsf{C}$
lin $f = u_1 ++ \ldots ++ u_n$

where f is a unique label, $(\mathsf{C}_1, \ldots, \mathsf{C}_m)$ are the nonterminals in the rule, and each u_i is either $\$_j$ (if $t_i = \mathsf{C}_j$) or s (if $t_i = s$).

That PGF is stronger than context-free grammars, is easily shown by the language $a^n b^n c^n$, whose grammar shows how **discontinuous constituents** are encoded as tuples of strings. The grammar is the following:

cat $\mathsf{S}; \mathsf{A}$
fun $s : (\mathsf{A}) \to \mathsf{S}; e : () \to \mathsf{A}; a : (\mathsf{A}) \to \mathsf{A}$
lincat $\mathsf{S} = \mathsf{Str}; \mathsf{A} = \mathsf{Str} * \mathsf{Str} * \mathsf{Str}$
lin $s = \$_1 ! 1 ++ \$_1 ! 2 ++ \$_1 ! 3$
$e = <[\,], [\,], [\,]>$
$a = <\text{“a”} ++ \$_1 ! 1, \text{“b”} ++ \$_1 ! 2, \text{“c”} ++ \$_1 ! 3>$

The general result about PGF is that it is equivalent to PMCFG (Parallel Multiple Context-Free Grammar, Seki et al. 1991). Hence any PGF grammar is parsable in polynomial time, with the exponent dependent on the grammar. This result was obtained for a more complex subset of GF by Ljunglöf (2004). Section 4 on parser generation will outline the result for PGF; it will also present a definition of PMCFG.

Being polynomial, PGF is not fully context-sensitive, but it is not mildly context-sensitive in the sense of (Joshi et al. 1991), because it does not have the **constant-growth property**. A counterexample is the following grammar, which defines the exponential language $\{a^{2^n} \mid n = 0, 1, 2, \ldots\}$:

cat S
fun $a : () \to \mathsf{S}; s : (\mathsf{S}) \to \mathsf{S}$
lincat $\mathsf{S} = \mathsf{Str}$
lin $a = \text{“a”}; s = \$_1 ++ \$_1$

3.2 Extensions of concrete syntax

A useful extension of concrete syntax is **free variation**, expressed by the operator |. Free variation is used in concrete syntax to indicate that different expressions make no semantic difference, that is, that they have the same abstract syntax. A term $t \mid u$ is well-formed in any type T, if both t and u have type T. In operational semantics, free variation requires the lifting of other operations to cover operands of the form $t|u$. For instance,

$$(t \mid u) \,!\, v = (t \,!\, v) \mid (u \,!\, v)$$

As shown by Ljunglöf (2004), the semantics can be given in such a way that the complexity of parsing PGF is not affected.

In practical applications of PGF, it is useful to have non-canonical forms that decrease the size of grammars by factoring out common parts. In particular, the use of **macros** factors out terms occurring in several linearization rules. The method of **common subexpression elimination** often results in a grammar whose code size is just one tenth of the original. This method is standardly applied as a back-end optimization in the GF grammar compiler (Section 5.4), which may otherwise result in code explosion.

3.3 Extensions of abstract syntax

One of the original motivations of GF was to implement type-theoretical semantics as presented in Ranta (1994). This kind of semantics requires that the abstract syntax notation has the strength of a **logical framework**, in the sense of Martin-Löf (1984) and Harper et al. (1993). What we have presented above is a special case of this, with three ingredients missing:

- **Dependent types**: a category may take trees as arguments.
- **Higher-order abstract syntax**: a function may take functions as arguments.
- **Semantic definitions**: trees may be given computation rules.

Ranta (2004) gives the typing rules and operational semantics for these extensions of GF. The extensions have been transferred to PGF as well, but only the Haskell implementation currently supports them. The reason for this is mainly that these extensions are rarely used in GF applications (Burke and Johannisson 2005 and Jonson 2006 being exceptions). Language-theoretically, their effect is either dramatic or null, depending on how the language defined by a grammar is conceived. If parsing is expected to return only well-typed trees, then dependent types together with semantic definitions makes parsing undecidable, because type checking is undecidable. If parsing and dependent type checking are separated, as they are in practical implementations, dependent types make no difference to the parsing complexity.

Metavariables are an extension useful for several purposes. In particular, they are used for encoding suppressed arguments when parsing grammars with erasing linearization rules (Section 4.5). In dependent type checking, they are used for unifying type dependencies, and in interactive syntax editors (Khegai et al. 2003), they are used for marking unfinished parts of syntax trees.

4 Parsing

PGF concrete syntax is simple but still too complex to be used directly for efficient parsing and for that reason it is converted to PMCFG. It is possible to do the conversion incrementally during the parsing but this slows down the process, so instead we do the conversion in the compiler. This means that we have duplicated information in the PGF file, because the two formalisms are equivalent, but this is a trade off between efficiency and grammar size. At the same time it is not feasible to use only the PMCFG because it might be quite big in some cases while the client might be interested only in linearization for which he or she can use only the PGF syntax.

4.1 PMCFG definition

A parallel multiple context-free grammar is a 8-tuple $G = (N, T, F, P, S, d, r, a)$ where:

- N is a finite set of categories and a positive integer $d(A)$ called dimension is given for each $A \in N$.
- T is a finite set of terminal symbols which is disjoint with N.
- F is a finite set of functions where the arity $a(f)$ and the dimensions $r(f)$ and $d_i(f)$ $(1 \leq i \leq a(f))$ are given for every $f \in F$. Let's for every positive integer d, $(T^*)^d$ denote the set of all d-tuples of strings over T. The function is a total mapping from $(T^*)^{d_1(f)} \times (T^*)^{d_2(f)} \times \cdots \times (T^*)^{d_{a(f)}(f)}$ to $(T^*)^{r(f)}$ and it is defined as:

  $f :=< \alpha_1, \alpha_2, \ldots, \alpha_{r(f)} >$

  Each α_i is a string of terminals and $\langle k; l \rangle$ pairs, where $1 \leq k \leq a(f)$ is called argument index and $1 \leq l \leq d_k(f)$ is called constituent index.
- P is a finite set of productions of the form:

  $A \to f[A_1, A_2, \ldots, A_{a(f)}]$

  where $A \in N$ is called result category, $A_1, A_2, \ldots, A_{a(f)} \in N$ are called argument categories and $f \in F$ is the function symbol. For the production to be well formed the conditions $d_i(f) = d(A_i)$ $(1 \leq i \leq a(f))$ and $r(f) = d(A)$ must hold.
- S is the start category and $d(S) = 1$.

$$S \to s[A]$$
$$A \to a[A]$$
$$A \to e[]$$

$$s := < \langle 1;1\rangle\ \langle 1;2\rangle\ \langle 1;3\rangle >$$
$$a := < a\ \langle 1;1\rangle, b\ \langle 1;2\rangle, c\ \langle 1;3\rangle >$$
$$e := < [], [], [] >$$

Figure 1. PMCFG for the $a^n b^n c^n$ language.

We use the same definition of PMCFG as is used by Seki and Kato (2008) and Seki et al. (1993) with the minor difference that they use variable names like x_{kl} while we use $\langle k;l\rangle$ to refer to the function arguments. In the actual implementation we also allow every category to be used as a start category. When the category has multiple constituents then they all are tried from the parser.

The $a^n b^n c^n$ language from section 3.1 is represented in PMCFG as shown in Figure 1. The dimension of S and A are $d(S) = 1$ and $d(A) = 3$. Functions s and e are with 0-arity and function a is with 1-arity i.e. $a(s) = 0$, $a(a) = 1$ and $a(e) = 0$.

4.2 PMCFG generation

Both PGF and PMCFG deal with tuples but PGF is allowed to have bounded integers and nested tuples while PMCFG is restricted to have only flat tuples containing only strings. To do the conversion we need to get rid of the integers and to flatten the tuples.

Let's take again the categories from section 2.4 as examples:

lincat $\mathsf{NP} = \mathsf{Str} * \mathsf{Int}_2 * \mathsf{Int}_3$
$\mathsf{VP} = (\mathsf{Str} * \mathsf{Str} * \mathsf{Str}) * (\mathsf{Str} * \mathsf{Str} * \mathsf{Str})$

The type for VP does not contain any integers so it is simply flattened to one tuple with six constituents. When the linearization type contains integers then the category is split into multiple PMCFG categories, one for each combination of integer values. The integers are removed and the remaining type is flattened to one tuple. In this case for NP we will have six categories and they will all have a single string as a linearization type. Category splitting is not an uncommon operation. For example, in part-of-speech tagging there are usually different tags for nouns in plural and nouns in singular. We do this also on a syntactic level and the NP category is split into:

NP_{11} NP_{12} NP_{13}
NP_{21} NP_{22} NP_{23}

Strings:

$$[] \Downarrow [] \quad "foo" \Downarrow "foo" \quad \frac{s \Downarrow v \;\; t \Downarrow w}{s ++ t \Downarrow v ++ w}$$

Bounded integers:

$$i \Downarrow i \; (i = 1, 2, \ldots)$$

Tuples:

$$\frac{t_1 \Downarrow v_1 \;\ldots\; t_n \Downarrow v_n}{< t_1, \ldots, t_n > \Downarrow < v_1, \ldots, v_n >}$$

Projections:

$$\frac{t \Downarrow < v_1, \ldots, v_n > \quad u \Downarrow i}{t!u \Downarrow v_i} \; i = 1, \ldots, n \qquad \frac{t \Downarrow \langle k; \pi \rangle \quad u \Downarrow i}{t!u \Downarrow \langle k; \pi \; i \rangle} \; i = 1, \ldots, n$$

Argument variable:

$$\$_k \Downarrow \langle k; \rangle$$

Parameter Substitution:

$$\frac{a_1 \ldots a_n \vdash t \Downarrow \langle k; \pi \rangle}{a_1 \ldots a_k \cup (\pi, i) \ldots a_n, \vdash t \Downarrow i} \quad \begin{array}{l} (\pi, m) \in T_p(A_k), \; i = 1, \ldots, m, \\ \forall j.((\pi, j) \in a_k \;\Rightarrow\; i = j) \end{array}$$

Table 3. Abstract interpretation of PGF.

The conversion from PGF rules to PMCFG productions starts with abstract interpretation (Table 3), which is very similar to the operational semantics in Table 2. The major difference is that the argument values are known only at run-time and actually the $\$_i$ terms should be replaced by $\langle i; l \rangle$ pairs in PMCFG for some l. We extend the PGF syntax with a $\langle i; \pi \rangle$ meta-value which is only used internally in the generation. Here, π is a sequence of indices and not only one index as it is the case in the final PMCFG. Like in the operational semantics, the rules define the relation $\gamma \vdash t \Downarrow v$ but this time the context γ is a sequence of assumption sets. Each assumption set a_k contains pairs (π, i) which say that if π is the sequence of indices $l_1 \ldots l_n$ then we assume that $\$_k!l_1 \ldots !l_n \Downarrow i$.

In addition, two sets have to be computed for each category C: $T_s(C)$ and $T_p(C)$. T_s is the set of all sequences of integers $i_1 \ldots i_n$ such that if x is an expression of type C then $x!i_1!i_2! \ldots !i_n$ is of type Str. T_p is the set of all (π, k) pairs where π is again a sequence $i_1 \ldots i_n$ but this time $x!i_1!i_2! \ldots !i_n$ is of type Int_k.

The abstract interpretation rules for strings, integers and tuples are exactly the same but the rule for argument variables is completely new. It says that since we do not know the actual value of the variable we just replace it with the meta-value $\langle k; \rangle$. Furthermore there is an additional rule for projection which deals with the case when we have argument variable on the left-hand side of a projection. Basically the rule for argument variables and the extra projection rule converts terms like $\$_k!l_1 \ldots !l_n$ to meta-values like $\langle k; \pi \rangle$ where

π is the sequence $l_1 \dots l_n$. Since the terms are well-typed at some point either $\pi \in T_s(A_k)$ or $(\pi, n) \in T_p(A_k)$ for some n will be the case. In the first case the meta value will be left unchanged in the evaluated term. The second case is more important because it triggers the parameter substitution rule. This rule is nondeterministic because it makes an arbitrary choice for i and records the choice in the context γ. The last side condition in the rule ensures that if we already had made some assumption for $\langle k; \pi \rangle$ we cannot choose any other value except the value that is already in the context. Since the integers are bounded we have only a finite set of choices which ensures the termination.

Let's use the linearization rule from section 2.4 as an example again:

$$\textbf{lin}\ \mathsf{Pred} = \$_1\ !\ 1 +\!\!+ (\$_2\ !\ (\$_1\ !\ 2))\ !\ (\$_1\ !\ 3)$$

The first subterm $\$_1\ !\ 1$ is simply reduced to $\langle 1; 1 \rangle$ by the derivation:

$$\frac{\$_1\ \Downarrow\ \langle 1; \rangle \quad 1\ \Downarrow\ 1}{\$_1!1\ \Downarrow\ \langle 1; 1 \rangle}$$

The derivation of the second subterm is more complex because it contains argument variables on the right hand side of a projection, so they have to be removed using the parameter substitution rule:

$$\frac{\dfrac{a_1\ a_2 \vdash \$_1 \Downarrow \langle 1; \rangle \quad a_1\ a_2 \vdash 2 \Downarrow 2}{a_1\ a_2 \vdash \$_1!2 \Downarrow \langle 1;2 \rangle}}{(a_1 \cup \{(2, x)\})\ a_2\ \vdash\ \$_1!2\ \Downarrow\ x}$$

Since the parameter substitution is nondeterministic there are two possible derivations but they differ only in the final value, so we use the variable x to denote either value 1 or 2. In a similar way we can deduce that $(a_1 \cup \{(2, x), (3, y)\})\ a_2\ \vdash\ \$_1!3\ \Downarrow\ y$, where y is either 1, 2 or 3. Combining the two results the derivation for the second term gives:

$$\frac{\dfrac{\$_2 \Downarrow \langle 2; \rangle \quad \$_1!2 \Downarrow x}{\$_2!(\$_1!2) \Downarrow \langle 2; x \rangle} \quad \$_1!3\ \Downarrow\ y}{(\$_2!(\$_1!2))!(\$_1!3)\ \Downarrow\ \langle 2; x\ y \rangle}$$

By applying the rule for the concatenation the final result we get:

$$\langle 1; 1 \rangle +\!\!+ \langle 2; x\ y \rangle$$

The output from the abstract interpretation can be converted directly to a PMCFG tuple. In well-typed terms a tuple can appear only at the top-level, inside another tuple or on the left-hand side of a record projection. In the abstract interpretation all tuples inside record projections are removed so that the only choice for the evaluated term is to be a tree of nested tuples with leaves of type either Str or Int_n. The tree strictly follows the structure of the linearization type of the result category. The term can be flattened just like we did with the linearization types.

For each possible tree a new unique function is generated with definition containing all leaves of type Str as tuple constituents. For the example above this will lead to six functions:

$$
\begin{aligned}
Pred_1 &:= \langle 1;1\rangle\ \langle 2;1\ 1\rangle \\
Pred_2 &:= \langle 1;1\rangle\ \langle 2;2\ 1\rangle \\
Pred_3 &:= \langle 1;1\rangle\ \langle 2;1\ 2\rangle \\
Pred_4 &:= \langle 1;1\rangle\ \langle 2;2\ 2\rangle \\
Pred_5 &:= \langle 1;1\rangle\ \langle 2;1\ 3\rangle \\
Pred_6 &:= \langle 1;1\rangle\ \langle 2;2\ 3\rangle
\end{aligned}
$$

The bounded integers in the term are used to determine the right PMCFG result category and each assumption set a_i in the context γ is used to determine the corresponding argument category in the production. One production is generated for every function:

$$
\begin{aligned}
S &\to Pred_1[\mathrm{NP}_{11}, \mathrm{VP}] \\
S &\to Pred_2[\mathrm{NP}_{21}, \mathrm{VP}] \\
S &\to Pred_3[\mathrm{NP}_{12}, \mathrm{VP}] \\
S &\to Pred_4[\mathrm{NP}_{22}, \mathrm{VP}] \\
S &\to Pred_5[\mathrm{NP}_{13}, \mathrm{VP}] \\
S &\to Pred_6[\mathrm{NP}_{23}, \mathrm{VP}]
\end{aligned}
$$

4.3 Common subexpression elimination in PMCFG

The produced PMCFG could be very big without some form of common subexpression elimination. There are three elimination techniques that are implemented so far and they are described in this section. All of them have been discovered in experiments with real grammars.

The first observation is that the conversion algorithm in the previous section always generates a pair of function definition and production rule using the same function. It happens to be pretty common that one function could be reused in two different productions because the original definitions are equivalent. In the real implementation first the definition is generated and after that it is compared with the already existing definitions. Only if it is distinct a new function is generated.

Another issue is that there are many constituents in the function bodies which are equal but are used in different places. For that reason we collect a list of distinct constituents and then the function definitions are rewritten to contain only the indices of the corresponding constituents.

The last observation is that there are groups of productions like:

$$
\begin{array}{cccc}
A \to f[B, C_1, D_1] & A \to f[B, C_2, D_1] & A \to f[B, C_3, D_1] & A \to f[B, C_4, D_1] \\
A \to f[B, C_1, D_2] & A \to f[B, C_2, D_2] & A \to f[B, C_3, D_2] & A \to f[B, C_4, D_2]
\end{array}
$$

Language	Productions			File Size (Kb)		
	Plain	Optimized	Ratio	Plain	Optimized	Ratio
Bulgarian	3629	3516	1.03	20359	5021	4.05
Danish	1696	1615	1.05	1399	593	2.36
English	1198	1165	1.03	1648	710	2.32
Finnish	-	141441	-	-	6357	-
German	11079	8078	1.37	56559	3027	18.68
Italian	-	1089621	-	-	106282	-
Norwegian	1773	1696	1.05	1418	596	2.38
Russian	5248	5077	1.03	7735	1261	6.13
Swedish	1535	1496	1.03	1161	577	2.01

Table 4. Grammar sizes in number of PMCFG productions and PGF file size, for the GF Resource Grammar Library.

where the list could be very large. This happens when in some linearization function some parameters are used only when another parameter has a specific value. The abstract interpretation could not detect all these cases. It detects only the parameters that are not used at all and then the conversion rules are introduced. The list of productions can be compressed by introducing extra conversion rules:

$$
\begin{aligned}
A &\to f[B, C, D] \\
C &\to \_[C_1] \\
C &\to \_[C_2] \\
C &\to \_[C_3] \\
C &\to \_[C_4] \\
D &\to \_[D_1] \\
D &\to \_[D_2]
\end{aligned}
$$

These optimizations have been implemented and tried with the resource grammar library (Ranta 2008), which is the largest collection of grammars written in GF. The produced grammar sizes are summarized in Table 4. The first column shows the grammar size in number of PMCFG productions and the second the total PGF file size.

The common subexpression optimization seems to reduce the file size from 2 to 18 times depending on the grammar. For two of the languages Finnish and Italian it is even impossible to compile the grammar without the optimization. The conversion was tried on a computer with 2 GB physical memory but it was not enough to fit the unoptimized grammar. The conversion for Italian is possible but currently it takes 2 days on a 2 GHz CPU. For that reason three other Romance languages are not listed: Catalan, French and Spanish. Their compilation should be possible but would probably require the same amount of time as Italian. The main problem is the compilation of the SlashVP rule. All Romance languages have clitic structures in the verb phrases which cause

exponential growth of the grammar size. The same applies to Interlingua which is an artificial language whose verb phrases also have clitics.

Some statistics were collected from the compiled Italian grammar which suggest directions for further optimizations. The production and function definitions generated from the SlashVP linearization function constitute 41% of the total grammar size. In the SlashVP size the dominant proportion is due to the PMCFG functions. There are 11280 functions where each has 321 constituents. Fortunately only 5% of these constituents are distinct. This suggests that the data is sparse and there should be a more compact representation for it. This also explains why the compilation is so slow. Each constituent is currently compiled independently and this means a lot of repeated work. If some kind of memoization were used, the compilation time could be reduced dramatically.

4.4 Parsing with PMCFG

Efficient recognition and parsing algorithms for MCFG have been described in (Nakanishi et al. 1997), Ljunglöf (2004) and (Burden and Ljunglöf 2005). MCFG is a linear form of PMCFG where each constituent of an argument is used exactly once. Ljunglöf (2004) gives an algorithm for approximating a PMCFG with an MCFG. With the approximation it is possible to use a parsing algorithm for MCFG to parse with a PMCFG, but after that a postprocessing step is needed to recover the original parse tree and to filter out spurious parse trees via unification. Instead we are using a parsing algorithm that works directly with PMCFG and also has the advantage that it is incremental. The incrementality is important because it can be used for word prediction (see section 5.1). The incremental algorithm itself will appear in a separate paper.

4.5 Parse trees

The output from the parser is a syntax tree or a set of trees where the nodes are labeled with PMCFG functions. The trees have to be converted back to PGF abstract expression. In the absence of high-order terms (section 3.3) the transformation is trivial. The definition of each PMCFG function is annotated with the corresponding PGF linearization function so they just have to be replaced. The PMCFG grammar might be erasing i.e. some argument might not be used at all. In this case the slot for this argument is filled with meta variable.

5 Using PGF

In this section, we will outline how PGF grammars can be used to construct natural language processing applications. We first list a number of operations that can be performed with a PGF grammar, along with some possible applications of these operations. We then give a brief overview of the APIs (Application Programmer's Interfaces) which are currently available for using PGF functionality in applications. Finally, we outline how PGF grammars can be produced

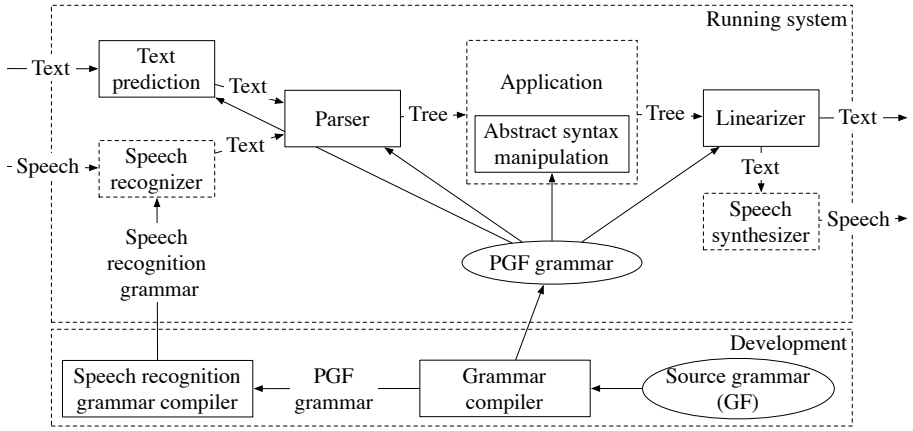

Figure 2. Overview of PGF applications.

from the more grammar-writer friendly format GF.

5.1 PGF operations

PGF grammars can be used for a wide range of tasks, either stand-alone, or as integral parts of a larger application. Figure 2 shows an overview of how PGF grammars can be used in natural language processing applications.

Parsing takes a string in some language and produces zero (in the case of an out-of-grammar input), one (in the the case of an unambiguous input), or more (for ambiguous inputs) abstract syntax trees. PGF parsing is for example useful for handling natural language user input in applications. This lets the rest of the application work on abstract syntax trees, which makes it easy to localize the application to new languages.

Text Prediction is related to parsing. The incremental PMCFG parsing algorithm can parse an incomplete input, and return the set of possible next tokens. If the last token in the given input is itself incomplete, the list of complete tokens can be given to the parser, and the result filtered to retain only those tokens that have the last (incomplete) token as a prefix.

Linearization, the production of text given an abstract syntax tree, can be used to produce natural language system outputs, either as text, or, by using speech synthesis software, speech output. In the latter case, the concrete syntax may contain annotations such as SSML tags which help the speech synthesizer.

Translation is the combination of parsing and linearization (Khegai 2006).

Abstract syntax generation, is made easy by the PGF abstract syntax type system. A potentially infinite lazy list of abstract syntax trees can be generated randomly, or exhaustively through iterative deepening. Random generation can be used to generate a monolingual corpus (generate a list of random abstract syntax trees and linearize them to given language), a multilingual parallel corpus (generate trees and linearize to several languages), or a treebank (generate trees,

```
data S  = Pred NP VP          abstract class S {...}
data NP = John                class Pred extends S { NP np; VP vp; ... }
data VP = Walk                abstract class NP {...}
                              class John extends NP {...}
                              abstract class VP {...}
                              class Walk extends VP {...}
```

Figure 3. Haskell and Java data types for the abstract syntax in Section 2.2.

```
readPGF :: FilePath → IO PGF
linearize :: PGF → Language → Tree → String
parse :: PGF → Language → Category → String → [Tree]
generateRandom :: PGF → Category → IO [Tree]
generateAll :: PGF → Category → [Tree]
complete :: PGF → Language → Category → String → [String]
```

Figure 4. A part of the Haskell PGF Interpreter API.

linearize, and output text, tree pairs).

Abstract syntax manipulation is useful in any application that analyzes or produces abstract syntax trees. The PGF type system makes it possible to expose abstract syntax manipulation to the user in a safe way. When statically typed languages such as Haskell or Java are used, it is possible to generate host language data types from an Figure 3 shows such data types generated from the example abstract syntax in Section 2.2.

5.2 PGF Interpreter API

A PGF Interpreter API allows an application programmer to make use of PGF grammars for the tasks listed above in a program written in some general purpose programming language. There are currently APIs for Haskell, Java, JavaScript, Prolog, C, C++, and web applications, with varying degrees of functionality. We examine the Haskell API in some detail. The other APIs, some of which are covered briefly below, have the same functionality, or subsets of it.

Haskell API Figure 4 shows the most important functions in the Haskell PGF Interpreter API. There are also functions for manipulating Tree value, for getting metadata about grammars, and variants of generation functions which give more control over the generation process, for example by setting a maximum tree height.

```
public class PGF {
  public static PGF readPGF (String path)
  public String linearize (String lang, Tree tree)
  public List<Tree> parse (String lang, String text)
}
```

Figure 5. A part of the Java PGF Interpreter API.

Java PGF Interpreter API The Java API is very similar to the Haskell API, but with a more object-oriented design, see Figure 5.

JavaScript PGF Interpreter API PGF grammars can also be converted into a JavaScript format, for which there is an interpreter that implements linearization, parsing and abstract syntax tree manipulation (Meza Moreno 2008).

PGF Interpreter Web Service PGF parsing, linearization and translation is also available as a web service, in the form of a FastCGI (Brown 1996) program with a RESTful (Fielding 2000) interface that returns JSON (Crockford 2006) structures. For example, a request to translate the sentence *this fish is fresh* from the concrete syntax `FoodEng` to all available concrete syntaxes may return the following JSON structure:

```
[{"from":"FoodEng","to":"FoodEng","text":"this fish is fresh"},
 {"from":"FoodEng","to":"FoodIta","text":"questo pesce è fresco"}]
```

5.3 Compiling PGF to other formats

The declarative nature of PGF makes it possible to translate PGF grammars to other grammar formalisms. It is theoretically interesting to produce algorithms for translating between different grammar formalisms. However, it also has practical applications, as it lets us use PGF grammars with existing software systems based on other grammar formalisms, for example speech recognizers (Bringert 2007a). When combined with PGF parsing, this makes it possible for an application to accept speech input, based solely on the information in the PGF grammar. Examples of such applications include dialogue systems (Ericsson et al. 2006; Bringert 2007b) and speech translation (Bringert 2008).

Most examples in the rest of this section will be based on the PGF grammar shown in Figure 7, which extends the earlier example grammar with the And and We functions. This grammar has been chosen to compactly include both agreement and left-recursion (at the expense of ambiguous parsing, this could be fixed with a slightly larger grammar).

```
cat S; NP; VP
fun And       : S → S → S
    Pred      : NP → VP → S
    John, We  : NP
    Walk      : VP
param Num  = Sg | Pl
param Pers = P1 | P2 | P3
lincat S   = Str
       NP  = {s : Str; n : Num; p : Pers}
       VP  = Num ⇒ Pers ⇒ Str
lin And x y = x ++ "und" ++ y
    Pred np vp = np.s ++ vp ! np.n ! np.p
    John = {s = "John"; n = Sg; p = P3}
    We   = {s = "wir"; n = Pl; p = P1}
    Walk =
      table {Sg ⇒ table {P1 ⇒ "gehe";
                         P2 ⇒ "gehst";
                         P3 ⇒ "geht"};
             Pl ⇒ table {P1 ⇒ "gehen";
                         P2 ⇒ "geht";
                         P3 ⇒ "gehen"}}
```

Figure 6. GF grammar.

```
cat S; NP; VP
fun And   : (S, S) → S
    Pred  : (NP, VP) → S
    John  : () → NP
    We    : () → NP
    Walk  : () → VP
lincat S  = Str
        NP = Str * Int2 * Int3
        VP = (Str * Str * Str)
             * (Str * Str * Str)
lin And  = $1 ++ "und" ++ $2
    Pred = $1 ! 1 ++ ($2 ! ($1 ! 2)) ! ($1 ! 3)
    John = <"John", 1, 3 >
    We   = <"wir", 2, 1 >
    Walk =
      < < "gehe",
          "gehst",
          "geht">,
        < "gehen",
          "geht",
          "gehen" > >
```

Figure 7. PGF grammar.

```
S   → S "und" S | NP1 VP2 | NP2 VP1
NP1 → "John"
NP2 → "wir"
VP1 → "gehen"
VP2 → "geht"
```

Figure 8. CFG.

```
S   → S "und" S{ And ($1, $2)}
    | NP1 VP2{ Pred ($1, $2)}
    | NP2 VP1{ Pred ($1, $2)}
NP1 → "John"{ John ()}
NP2 → "wir"{ We ()}
VP1 → "gehen"{ Walk ()}
VP2 → "geht"{ Walk ()}
```

Figure 9. Fig. 8 with interpretation.

$$
\begin{array}{ll}
\mathsf{S} & \to \mathsf{NP}_1\ \mathsf{S}_2 \mid \mathsf{NP}_2\ \mathsf{S}_3 \\
\mathsf{S}_2 & \to \mathsf{VP}_2 \mid \mathsf{VP}_2\ \mathsf{S}_4 \\
\mathsf{S}_3 & \to \mathsf{VP}_1 \mid \mathsf{VP}_1\ \mathsf{S}_4 \\
\mathsf{S}_4 & \to \text{“und”}\ \mathsf{S} \mid \text{“und”}\ \mathsf{S}\ \mathsf{S}_4 \\
\mathsf{NP}_1 & \to \text{“John”} \\
\mathsf{NP}_2 & \to \text{“wir”} \\
\mathsf{VP}_1 & \to \text{“gehen”} \\
\mathsf{VP}_2 & \to \text{“geht”}
\end{array}
$$

Figure 10. Non-left-recursive CFG.

$$
\begin{array}{ll}
\mathsf{S} & \to\ \mathsf{NP}_1\ \mathsf{S}_2\{\ \$_2\ (\$_1)\ \} \\
 & \quad \mid \mathsf{NP}_2\ \mathsf{S}_3\{\ \$_2\ (\$_1)\ \} \\
\mathsf{S}_2 & \to\ \mathsf{VP}_2\{\ \lambda x.\mathsf{Pred}\ (x, \$_1)\ \} \\
 & \quad \mid \mathsf{VP}_2\ \mathsf{S}_4\{\ \lambda x.\$_2\ (\mathsf{Pred}\ (x, \$_1))\ \} \\
\mathsf{S}_3 & \to\ \mathsf{VP}_1\{\ \lambda x.\mathsf{Pred}\ (x, \$_1)\ \} \\
 & \quad \mid \mathsf{VP}_1\ \mathsf{S}_4\{\ \lambda x.\$_2\ (\mathsf{Pred}\ (x, \$_1))\ \} \\
\mathsf{S}_4 & \to\ \text{“und”}\ \mathsf{S}\{\ \lambda x.\mathsf{And}\ (x, \$_1)\ \} \\
 & \quad \mid \text{“und”}\ \mathsf{S}\ \mathsf{S}_4\{\ \lambda x.\$_2\ (\mathsf{And}\ (x, \$_1))\ \} \\
\mathsf{NP}_1 & \to \text{“John”}\{\ \mathsf{John}\ ()\ \} \\
\mathsf{NP}_2 & \to \text{“wir”}\{\ \mathsf{We}\ ()\ \} \\
\mathsf{VP}_1 & \to \text{“gehen”}\{\ \mathsf{Walk}\ ()\ \} \\
\mathsf{VP}_2 & \to \text{“geht”}\{\ \mathsf{Walk}\ ()\ \}
\end{array}
$$

Figure 11. Fig. 10 with interpretation.

$$
\begin{array}{ll}
\mathsf{S} & \to \mathsf{NP}_1\ \mathsf{VP}_2\ \mathsf{S}_2 \mid \mathsf{NP}_2\ \mathsf{VP}_1\ \mathsf{S}_2 \\
\mathsf{S}_2 & \to \text{“und”}\ \mathsf{S} \mid \epsilon \\
\mathsf{NP}_1 & \to \text{“John”} \\
\mathsf{NP}_2 & \to \text{“wir”} \\
\mathsf{VP}_1 & \to \text{“gehen”} \\
\mathsf{VP}_2 & \to \text{“geht”}
\end{array}
$$

Figure 12. Regular grammar.

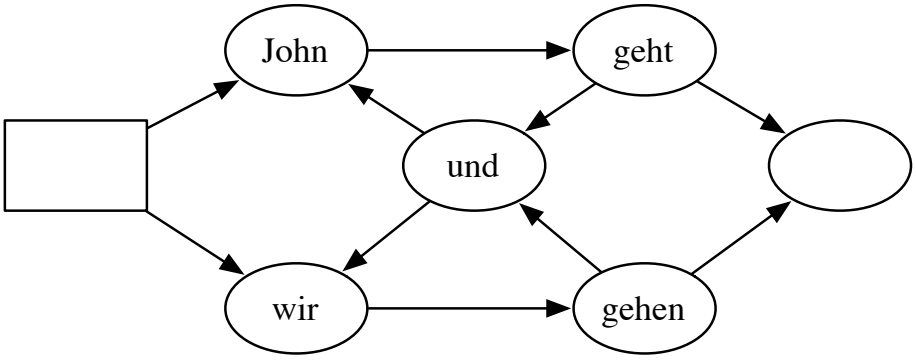

Figure 13. Finite-state automaton.

$\mathsf{S} \rightarrow \mathsf{A}_0\ \mathsf{A}_1\ \mathsf{A}_2$
$\mathsf{A} \rightarrow \mathsf{A}_0 \mid \mathsf{A}_1 \mid \mathsf{A}_2$
$\mathsf{A}_0 \rightarrow \epsilon \mid$ "a" A_0
$\mathsf{A}_1 \rightarrow \epsilon \mid$ "b" A_1
$\mathsf{A}_2 \rightarrow \epsilon \mid$ "c" A_2

Figure 14. Context-free approximation of the PMCFG grammar for the $a^n b^n c^n$ language.

Context-free approximation

A context-free grammar (CFG) that approximates a PGF grammar can be produced by first producing a PMCFG as described earlier. The PMCFG is then approximated with a CFG by converting each PMCFG category-field pair to a CFG category. In the general case, this is an approximation, since PMCFG is a stronger formalism than CFG. The example language with agreement is converted to the CFG shown in Figure 8.

The PMCFG grammar given in Section 4.1 for the context-sensitive language $a^n b^n c^n$ is approximated by the context-free grammar shown in Figure 14. In this case the context-free approximation is overgenerating, as it generates the language $a^* b^* c^*$. The simpler $a^n b^n$ language is context-free, but may be described by either a context-sensitive grammar in a similar way to the $a^n b^n c^n$ language above, or by a context-free grammar. The context-free approximation will preserve the language described by an already context-free grammar, but not necessarily the language of a context-sensitive grammar that defines a context-free language. It is not possible to devise an algorithm that converts all context-sensitive grammars that generate context-free languages to context-free grammars, since deciding whether a context-sensitive language is context-free is undecidable. We conjecture that this also holds for deciding whether a PMCFG generates a context-free language.

Context-free transformations

Our PGF compiler also implements a number of transformations on context-free grammars, such as cycle elimination, bottom-up and top-down filtering, left-recursion elimination, identical category elimination, and EBNF compaction. This makes it possible to produce grammars in a number of restricted context-free formats as required by speech recognition software.

Embedded tree building code

The rules for producing abstract syntax trees can be preserved through the grammar translations listed above. This makes it possible to include abstract syntax tree building code in the generated context-free grammars, for example in SISR (Burke and Van Tichelen 2006) format. Figure 9 shows such an annotated grammar. As is shown in Figure 11, the abstract syntax tree building annotations can be carried through the left-recursion removal transformation, by the use of λ-terms.

Regular / finite-state approximation

The PGF grammar compiler can also approximate the produced context-free grammars with regular grammars, which can in turn be compiled to finite-state automata. An example of this is shown in Figures 12 and 13. This lets us support speech recognizers which require regular or finite-state language models, or for typical finite-state natural language processing tasks such as marking noun phrases in a corpus.

5.4 Compiling GF to PGF

While PGF is suitable for efficient and simple implementation, PGF grammars are not meant to be produced by humans. Rather, it is intended as the target language of a compiler from a high-level grammar language. In this section, we will outline the differences between the human-friendly GF language, and the machine-friendly PGF language, and how grammars written in GF can be translated to PGF. The PGF syntax and semantics can be seen as a subset of the GF syntax and semantics, while the PGF type system is less strict than that of GF. Where the GF type system is concerned with correctness, the PGF type system only ensures safety.

Tables and records

In GF, there are two separate constructs that correspond to PGF tuples: tables and records. In GF, tables are tuples whose values all have the same type, and finite algebraic data types or records are used to select values from tables. Tables are used to implemented parametric features such as inflection.

Records, on the other hand, can have values of different type, but the selectors used with records are labels which must be known statically. Records are used to implement inherent features and discontinuous constituents.

Both tables and records can be nested, but they always have a statically known size, which only depends on their type. This makes it possible to translate both records and tables to PGF tuples. For example, the GF grammar shown in Figure 6 is translated to the PGF grammar in Figure 7.

Structured parameter values and labels

As noted above, in GF, table projection is done with structured values known as parameters. These can be combinations of non-recursive algebraic data types and records of parameters. Since parameter records contain a known number of values, and the algebraic parameter values are non-recursive, each parameter type contains a finite number of values. This makes it possible to translate each parameter type to a bounded integer type, suitable for projection on PGF tuples.

Stricter type checking

As noted above, the GF concrete syntax type system is stricter than the PGF type system. In PGF, bounded integers are used to represent all parameter types and record labels, which means that many distinctions made by the GF type checker are not available in PGF. The differences between the GF and PGF type system can be compared to the differences between the Java type system and JVM bytecode verification. The type system for abstract syntax is identical in GF and PGF.

Pattern matching

Table projection in GF can be done by pattern matching with rather complex patterns. When compiling to PGF, all tables are expanded to have one branch with for each possible parameter value, to allow for translation to PGF tuples.

Modularity

GF grammars are organized into modules which can be compiled to core GF separately, like Java classes or C object files. PGF on the other hand has no module system and is a single file, similar to a statically linked executable. This simplifies PGF implementations and makes it easy to distribute PGF grammars.

Auxiliary operations

In GF, auxiliary operations can be defined in order to implement for example morphological paradigms or common syntactic operations. These operations, like all GF concrete syntax can include complex features such as higher-order functions and pattern matching on strings. During the translation to PGF, all

auxiliary operations are inlined, and all expressions are evaluated to produce valid PGF terms. This means, for example, that all strings which are used in pattern matching or in-token concatenation must be statically computable.

Because this inlining and computation can produce very large and repetitive PGF terms, common sub-expression elimination is performed. This can recover some of the auxiliary operations, but in a simpler form, but it can also discover new opportunities for code sharing, as PGF macros allow open terms, are type agnostic, and shared between code that comes from unrelated GF modules.

6 Results and evaluation

6.1 Systems using PGF

A number of natural language applications have been implemented using PGF or its predecessor GFCM. Above, we have listed a number of individual tasks that can be performed with PGF grammars. However, realistic applications often make use PGF for more than one task. This helps avoid the duplicated work involved in manually implementing multiple components which cover the same language fragment.

GOTTIS (Bringert et al. 2005) is an experimental multimodal and multilingual dialogue system for public transportation queries. It uses the a generated Nuance GSL speech recognition grammar for speech input, embedded parsing and linearization for system input and output, and generated Java data types for analysing input abstract syntax trees and producing output abstract syntax trees. Both input and output in GOTTIS make use of *multimodal* grammars. The input grammar allows *integrated multimodality*, e.g. "I want to go here", accompanied by a click on a map. This is implemented by using a two-field record (PGF tuple) to represent spatial expressions, where one field contains the spoken component, and another contains the click component. System output makes use of *parallel multimodality*; one concrete syntax produces spoken route descriptions, and another produces drawing instructions which are used to display routes on a map. Other examples of using GF grammars for formal languages include the WebALT mathematics problem editor (Cohen et al. 2006) and the KeY software specification editor (Burke and Johannisson 2005),

PGF abstract syntax can be used as a specification of a dialogue manager (Ranta and Cooper 2004). Together with the existing speech recognition grammar generation with embedded semantic interpretation tags, and the PGF to JavaScript compiler, this can be used to generate complete multimodal and multilingual VoiceXML dialogue systems (Bringert 2007b).

DUDE (Lemon and Liu 2006) and its extension REALL-DUDE (Lemon et al. 2006) are environments where non-experts can develop dialogue systems based on Business Process Models describing the applications. From keywords, prompts and answer sets defined by the developer, the system generates a GF grammar. This grammar is used for parsing input, and for generating a language model in SLF or GSL format.

Several syntax-directed editors for controlled languages (Khegai et al. 2003; Johannisson et al. 2003; Meza Moreno and Bringert 2008), have been implemented using PGF and its predecessors. They make use of abstract syntax manipulation, parsing linearization.

PGF can be used to implement complete limited domain speech translation systems that use PGF to produce speech recognition grammars and to perform parsing and linearization (Bringert 2008).

Text prediction can be used to implement text-based user interfaces which can show the user what inputs are allowed by the grammar. Examples of applications where this might be useful are editors for controlled languages, language learning software or limited domain translation software such as tourist phrase books. A web-based controlled language translator using text prediction is currently being developed.

Jonson (2006, 2007) used random corpus generation to produce statistical language models (SLM) for speech recognition from a domain-specific grammar. When combined with a general SLM trained on existing general domain text, the precision on in-grammar inputs was largely unchanged, while out-of-grammar inputs where handled with much higher precision, compared to a pure grammar-based language model. When producing corpora for training SLMs, dependently typed abstract syntax can be used to reduce over-generation (Jonson 2006). Other, as yet unexplored, applications of PGF abstract syntax generation are the use of generated multilingual parallel corpora for training statistical machine translation systems, and the use of generated treebanks for training statistical parsers.

The grammars from the GF resource grammar library (Ranta 2008) can be used not only to implement application-specific grammars, but also as general wide-coverage grammars. For example, the English resource grammar, with a large lexicon and some minor syntax extensions, covers a significant portion of the sentences in the FraCaS semantic test suite (Cooper et al. 1996).

7 Related work

Compilation of grammars to low-level formats is an old idea. The most well-known examples are parser generators in the YACC family (Johnson 1975). The output of YACC-like parser generation is a piece of host language source code, which can be seamlessly integrated in a program written in the host language. YACC-like systems for full context-free grammars suitable for natural language include the work by Tomita and Carbonell (1987), the NLYACC system (Ishii et al. 1994), and the GLR extension of the Happy parser tool for Haskell (Callaghan and Medlock 2004). The main differences between PGF and the YACC family are that PGF is stronger than context-free grammars, that PGF contains no host language code and is therefore portable to many host languages, that PGF supports linearization in addition to parsing, and that PGF grammars can be multilingual.

HPSG (Pollard and Sag 1994) and LFG (Bresnan 1982) are grammar for-

malisms used for large-scale grammar writing and processing. In their implementation, the use of optimizing compilers is essential, to support at the same time high-level grammar writing and efficient run-time execution. HPSG compilers, for instance, have used advanced compiler techniques such as data flow analysis (Minnen et al. 1995). The main focus in both grammars is parsing, but also generation is supported. Both in HPSG and LFG, systems of parallel grammars for different languages have been developed, but neither formalism is multilingual in the way GF/PGF is. The currently most popular implementations are LKB (Copestake 2002) for HPSG and XLE (Kaplan and Maxwell 2007) for LFG. To our knowledge, neither formalism supports generation of portable low-level code.

An emerging species of embedded grammar applications is language models for speech recognition. Regulus (Rayner et al. 2006) is a system that compiles high-level feature-based grammars into low-level context-free grammars in the Nuance format (Nua 2003). PGF can likewise be compiled into Nuance and a host of other speech grammar formats (Bringert 2007a).

Type-theoretical grammar formats based on Curry's distinction between tectogrammar and phenogrammar have gained popularity in the recent years: ACG (Abstract Categorial Grammars) (de Groote 2001), HOG (Higher-Order Grammars) (Pollard 2004), and Lambda Grammars (Muskens 2001) are the most well-known examples besides GF. The work in implementing these formalisms has not come so far as in GF. One obvious idea for implementing them is to use compilation to PGF. But it remains to be seen if PGF is general enough to support this. For instance, ACG is more general than PGF in the sense that linearization types can be function types, but less general in the sense that functions have to be linear (that is, use every argument exactly once). This means that the style of defining functions is very different, for instance, that a rule written with multiple occurrences of a variable in PGF is in ACG encoded as a higher-order function.

PMCFG, while known for almost two decades and having nice computational properties, has not been used for practical grammar writing. Even the use of PMCFG as a target format for grammar compilation seems to be new to the GF project.

8 Conclusion

PGF was first created as a low-level target format for compiling high-level grammars written in GF. The division between high-level source formats and low-level target formats has known advantages in programming language design, which have been confirmed in the case of GF and PGF. One distinguishing property is redundancy: the absence of redundancy from PGF makes is maximally easy to write PGF interpreters, to compile PGF to other formats, and to reason about PGF. In GF, on the other hand, computationally redundant features, such as intensional type distinctions, inlinable functions, and separately compilable modules, support the programmer's work by permitting useful er-

ror messages and an abstract programming style. GF as compiled to PGF has made it possible to build grammar-based systems that combine linguistic coverage (the GF resource grammar library) with efficient run-time behaviour (mostly linear-time generation, incremental PMCFG parsing) and integration with other system components (web pages via JavaScript, spoken language models via context-free approximations). For other grammar formalisms than GF, compilation to PGF could be used as an economical implementation technique, which would moreover make it possible to link together grammars written in different high-level formalisms.

References

Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, December 1997. ISBN 0521582741.

J. Bresnan, editor. *The Mental Representation of Grammatical Relations*. MIT Press, 1982.

Björn Bringert. Speech Recognition Grammar Compilation in Grammatical Framework. In *Proceedings of the Workshop on Grammar-Based Approaches to Spoken Language Processing, Prague, Czech Republic*, pages 1–8. Association for Computational Linguistics, June 2007a. URL `http://www.aclweb.org/anthology/W/W07/W07-1801`.

Björn Bringert. Rapid Development of Dialogue Systems by Grammar Compilation. In Simon Keizer, Harry Bunt, and Tim Paek, editors, *Proceedings of the 8th SIGdial Workshop on Discourse and Dialogue, Antwerp, Belgium*, pages 223–226. Association for Computational Linguistics, September 2007b. URL `http://www.sigdial.org/workshops/workshop8/Proceedings/SIGdial39.pdf`.

Björn Bringert. Speech Translation with Grammatical Framework. In *Coling 2008: Proceedings of the workshop on Speech Processing for Safety Critical Translation and Pervasive Applications, Manchester, UK*, pages 5–8. Coling 2008 Organizing Committee, August 2008. URL `http://www.cs.chalmers.se/~bringert/publ/gf-slt/gf-slt.pdf`.

Björn Bringert, Robin Cooper, Peter Ljunglöf, and Aarne Ranta. Multimodal Dialogue System Grammars. In *Proceedings of DIALOR'05, Ninth Workshop on the Semantics and Pragmatics of Dialogue, Nancy, France*, pages 53–60, June 2005. URL `http://dialor05.loria.fr/Papers/07-BjornBringert.pdf`.

Mark R. Brown. FastCGI: A High-Performance Gateway Interface. In Anton Eliëns, editor, *Programming the Web - a search for APIs, Fifth International World Wide Web Conference (WWW5), Paris, France*, May 1996. URL `http://www.cs.vu.nl/~eliens/WWW5/papers/FastCGI.html`.

Håkan Burden and Peter Ljunglöf. Parsing Linear Context-Free Rewriting Systems. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 11–17, Vancouver, British Columbia, 2005. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W05/W05-1502`.

David Burke and Luc Van Tichelen. Semantic Interpretation for Speech Recognition (SISR) Version 1.0. Working draft, W3C, November 2006. URL `http://www.w3.org/TR/2006/WD-semantic-interpretation-20061103`.

David A. Burke and Kristofer Johannisson. Translating Formal Software Specifications to Natural Language. In *Logical Aspects of Computational Linguistics*, volume 3492 of *Lecture Notes in Computer Science*, pages 51–66. Springer, Heidelberg, May 2005. doi: 10.1007/11422532_4.

P. Callaghan and B. Medlock. Happy-GLR, 2004. URL `http://www.dur.ac.uk/p.c.callaghan/happy-glr/`.

Arjeh Cohen, Hans Cuypers, Karin Poels, Mark Spanbroek, and Rikko Verrijzer. WExEd - WebALT Exercise Editor for Multilingual Mathematical Exercises. In Mika Seppälä, Sebastian Xambo, and Olga Caprotti, editors, *WebALT 2006, First WebALT Conference and Exhibition, Eindhoven, The Netherlands*, pages 141–145, January 2006. URL `http://www.win.tue.nl/~amc/pub/wexed.pdf`.

Robin Cooper, Dick Crouch, Jan van Eijck, Chris Fox, Josef van Genabith, Jan Jaspars, Hans Kamp, David Milward, Manfred Pinkal, Massimo Poesio, Steve Pulman, Ted Briscoe, Holger Maier, and Karsten Konrad. A Semantic Test Suite. In *Using the Framework, Deliverable D16*, chapter 3. The FRACAS Consortium, January 1996. URL `ftp://ftp.cogsci.ed.ac.uk/pub/FRACAS/del16.ps.gz`.

A. Copestake. *Implementing Typed Feature Structure Grammars*. CSLI Publications, 2002.

Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006. URL `http://www.ietf.org/rfc/rfc4627.txt`.

Haskell B. Curry. Some Logical Aspects of Grammatical Structure. In Roman O. Jakobson, editor, *Structure of Language and its Mathematical Aspects, volume 12 of Symposia on Applied Mathematics*, pages 56–68. American Mathematical Society, Providence, 1961.

Philippe de Groote. Towards abstract categorial grammars. In *Proceedings of 39th Annual Meeting of the Association for Computational Linguistics, Toulouse, France*, pages 252–259, Morristown, NJ, USA, July 2001. Association for Computational Linguistics. doi: 10.3115/1073012.1073045.

Stina Ericsson, Gabriel Amores, Björn Bringert, Håkan Burden, Ann-Charlotte Forslund, David Hjelm, Rebecca Jonson, Staffan Larsson, Peter Ljunglöf, Pilar Manchón, David Milward, Guillermo Pérez, and Mikael Sandin. Software illustrating a unified approach to multimodality and multilinguality in the in-home domain. deliverable 1.6, 2006. URL `http://www.talk-project.org/fileadmin/talk/publications_public/deliverables_public/D1_6.pdf`.

Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. URL `http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf`.

R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *JACM*, 40(1):143–184, 1993.

Masayuki Ishii, Kazuhisa Ohta, and Hiroaki Saito. An efficient parser generator for natural language. In *Proceedings of the 15th conference on Computational linguistics*, pages 417–420, Morristown, NJ, USA, 1994. Association for Computational Linguistics. doi: 10.3115/991886.991959.

Kristofer Johannisson, Janna Khegai, Markus Forsberg, and Aarne Ranta. From Grammars to Gramlets. In *The Joint Winter Meeting of Computing Science and Computer Engineering*. Chalmers University of Technology, 2003.

S. C. Johnson. Yacc — yet another compiler compiler. Technical Report CSTR-32, AT & T Bell Laboratories, Murray Hill, NJ, 1975.

Rebecca Jonson. Generating Statistical Language Models from Interpretation Grammars in Dialogue Systems. In *EACL 2006, 11st Conference of the European Chapter of the Association for Computational Linguistics*, 2006. URL `http://acl.ldc.upenn.edu/E/E06/E06-1008.pdf`.

Rebecca Jonson. Grammar-based context-specific statistical language modelling. In *Proceedings of the Workshop on Grammar-Based Approaches to Spoken Language Processing*, pages 25–32, Prague, Czech Republic, June 2007. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W07/W07-1804`.

A. Joshi, K. Vijay-Shanker, and D. Weir. The convergence of mildly context-sensitive grammar formalisms. In P. Sells, S. Shieber, and T. Wasow, editors, *Foundational Issues in Natural Language Processing*, pages 31–81. MIT Press, 1991.

R. Kaplan and J. Maxwell. XLE Project Homepage, 2007. URL `http://www2.parc.com/isl/groups/nltt/xle/`.

Janna Khegai. Grammatical Framework (GF) for MT in sublanguage domains. In *Proceedings of EAMT-2006, 11th Annual conference of the European Association for Machine Translation, Oslo, Norway*, pages 95–104, June 2006. URL `http://www.mt-archive.info/EAMT-2006-Khegai.pdf`.

Janna Khegai, Bengt Nordström, and Aarne Ranta. Multilingual Syntax Editing in GF. In Alexander Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing*, volume 2588 of *Lecture Notes in Computer Science*, pages 199–204. 2003. doi: 10.1007/3-540-36456-0_48.

Oliver Lemon and Xingkun Liu. DUDE: a Dialogue and Understanding Development Environment, mapping Business Process Models to Information State Update dialogue systems. In *EACL 2006, 11st Conference of the European Chapter of the Association for Computational Linguistics*, 2006. URL `http://www.aclweb.org/anthology-new/E/E06/E06-2004.pdf`.

Oliver Lemon, Xingkun Liu, Daniel Shapiro, and Carl Tollander. Hierarchical Reinforcement Learning of Dialogue Policies in a development environment for dialogue systems: REALL-DUDE. In *BRANDIAL'06, Proceedings of the 10th Workshop on the Semantics and Pragmatics of Dialogue*, pages 185–186, September 2006. URL `http://www.ling.uni-potsdam.de/brandial/Proceedings/brandial06_lemon_etal.pdf`.

Peter Ljunglöf. *Expressivity and Complexity of the Grammatical Framework*. PhD thesis, Göteborg University, Göteborg, Sweden, 2004. URL `http://www.ling.gu.se/~peb/pubs/p04-PhD-thesis.pdf`.

Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984.

J. McCarthy. Towards a mathematical science of computation. In *Proceedings of the Information Processing Cong. 62*, pages 21–28, Munich, West Germany, August 1962. North-Holland.

Moisés S. Meza Moreno. Implementation of a JavaScript Syntax Editor and Parser for Grammatical Framework. Master's thesis, Chalmers University of Technology, 2008.

Moisés S. Meza Moreno and Björn Bringert. Interactive Multilingual Web Applications with Grammatical Framework. In Bengt Nordström and Aarne Ranta, editors, *Advances in Natural Language Processing, 6th International Conference, GoTAL 2008, Gothenburg, Sweden*, volume 5221 of *Lecture Notes in Computer Science*, pages 336–347, Heidelberg, August 2008. Springer. doi: 10.1007/978-3-540-85287-2_32.

G. Minnen, D. Gerdemann, and T. Gotz. Off-line optimization for earleystyle hpsg processing, 1995. URL `citeseer.ist.psu.edu/article/minnen95offline.html`.

R. Montague. *Formal Philosophy*. Yale University Press, New Haven, 1974. Collected papers edited by Richmond Thomason.

R. Muskens. Lambda Grammars and the Syntax-Semantics Interface. In R. van Rooy and M. Stokhof, editors, *Proceedings of the Thirteenth Amsterdam Colloquium*, pages 150–155, Amsterdam, 2001.

Ryuichi Nakanishi, Keita Takada, and Hiroyuki Seki. An Efficient Recognition Algorithm for Multiple Context-Free Languages. In *Fifth Meeting on Mathematics of Language*. The Association for Mathematics of Language, August 1997. URL `http://citeseer.ist.psu.edu/65591.html`.

Nuance Speech Recognition System 8.5: Grammar Developer's Guide. Nuance Communications, Inc., Menlo Park, CA, USA, December 2003.

C. Pollard and I. Sag. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, 1994.

Carl Pollard. Higher-Order Categorial Grammar. In *Proceedings of Categorial Grammars 2004*, pages 340–361, June 2004. URL `http://www.ling.ohio-state.edu/~hana/hog/pollard2004-CG.pdf`.

A. Ranta. GF Resource Grammar Library, 2008. URL `http://digitalgrammars.com/gf/lib/`.

A. Ranta. *Type Theoretical Grammar*. Oxford University Press, 1994.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004. ISSN 0956-7968. doi: 10.1017/S0956796803004738.

Aarne Ranta and Robin Cooper. Dialogue Systems as Proof Editors. *Journal of Logic, Language and Information*, 13(2):225–240, 2004. ISSN 0925-8531. doi: 10.1023/B:JLLI.0000024736.34644.48.

Manny Rayner, Beth A. Hockey, and Pierrette Bouillon. *Putting Linguistics into Speech Recognition: The Regulus Grammar Compiler*. CSLI Publications, Ventura Hall, Stanford University, Stanford, CA 94305, USA, July 2006. ISBN 1575865262.

Hiroyuki Seki and Yuki Kato. On the Generative Power of Multiple Context-Free Grammars and Macro Grammars. *IEICE-Transactions on Info and Systems*, E91-D(2):209–221, 2008. doi: 10.1093/ietisy/e91-d.2.209.

Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88(2):191–229, October 1991. ISSN 0304-3975. doi: 10.1016/0304-3975(91)90374-B.

Hiroyuki Seki, Ryuichi Nakanishi, Yuichi Kaji, Sachiko Ando, and Tadao Kasami. Parallel Multiple Context-Free Grammars, Finite-State Translation Systems, and Polynomial-Time Recognizable Subclasses of Lexical-Functional Grammars. In *31st Annual Meeting of the Association for Computational Linguistics*, pages 130–140. Ohio State University, Association for Computational Linguistics, June 1993. doi: 10.3115/981574.981592.

Masaru Tomita and Jaime G. Carbonell. The universal parser architecture for knowledge-based machine translation. In *IJCAI*, pages 718–721, 1987.

Paper VII

A Pattern for Almost Compositional Functions

Journal of Functional Programming, and ICFP 2006, Portland

A Pattern for Almost Compositional Functions

Björn Bringert and Aarne Ranta
Department of Computer Science and Engineering
Chalmers University of Technology
and University of Gothenburg
SE-412 96 Göteborg, Sweden
{bringert,aarne}@chalmers.se

Abstract

This paper introduces a pattern for almost compositional functions over recursive data types, and over families of mutually recursive data types. Here "almost compositional" means that for all of the constructors in the type(s), except a limited number of them, the result of the function depends only on the constructor and the results of calling the function on the constructor's arguments. The pattern consists of a generic part constructed once for each data type or family of data types, and a task-specific part. The generic part contains the code for the predictable compositional cases, leaving the interesting work to the task-specific part. Examples of the pattern are given, implemented in dependent type theory with inductive families, in Haskell with generalized algebraic data types and rank-2 polymorphism, and in Java using a variant of the Visitor design pattern. The relationships to the "Scrap Your Boilerplate" approach to generic programming, and to general tree types in dependent type theory, are investigated by reimplementing our operations using those frameworks.

1 Introduction

This paper addresses the issue of repetitive code in operations on rich data structures. To give concrete examples of what we would like to be able to do, we start by giving some motivating problems.

1.1 Some motivating problems

Suppose that you have an abstract syntax definition with many syntactic types such as statement, expression, and variable.

1. Write a function that prepends an underscore to the names of all variables in a program. Do this with a case expression that has just two branches: one for the variables, and another for the rest.

2. Write a function that gives unique names to all variables in a program. Use only three cases: variable bindings, variable uses, and the rest.

3. Write a function that constructs a symbol table containing all variables declared in a program, and the type of each variable. Do this with only two cases: one for declarations, another for the rest.

4. Write a function that replaces increment statements with the corresponding assignments. Use only two cases: one for increments, and one for the rest.

One problem when writing recursive functions which need to traverse rich data structures is that the straightforward way to write them involves large amounts of traversal code which tends to be repeated in each function. There are several problems with this:

- The repeated traversals are probably implemented using copy-and-paste or retyping, both of which are error-prone and can lead to maintenance problems.
- When we add a constructor to the data type, we need to change all functions that traverse the data type, many of which may not need any specific behavior for the new constructor.
- Repeated traversal code obscures the interesting cases where the functions do their real work.
- The need for complete traversal code for the whole family of data types in every function can encourage a less modular programming style where multiple operations are collected in a single function.

1.2 The solution

The pattern which we present in this paper allows the programmer to solve problems such as the above in a (hopefully) intuitive way. First we write the traversal code once and for all for our data type or family of data types. We then reuse this component to succinctly express the operations which we want to define.

1.3 Article overview

We first present the simple case of a single recursive algebraic data type, and show examples of using the pattern in plain Haskell 98 (Peyton Jones 2003a). After that, we generalize this to the more complex case of a family of data types, and show how the pattern can be used in dependent type theory (Martin-Löf 1984; Nordström et al. 1990) with inductive families (Dybjer 1994) and in Haskell with generalized algebraic data types (Peyton Jones et al. 2006; Augustsson and Petersson 1994) and rank-2 polymorphism (Leivant 1983; Peyton Jones

et al. 2007). We then prove some properties of our compositional operations, using the laws for applicative functors (McBride and Paterson 2008). We go on to express the pattern in Java (Gosling et al. 2005), using a variant of the Visitor design pattern (Gamma et al. 1995). We also briefly describe some tools which can be used to automate the process of writing the necessary support code for a given data type. Finally, we discuss some related work in generic programming, type theory, object-oriented programming and compiler construction, and provide some conclusions.

2 Abstract Syntax and Algebraic Data Types

Algebraic data types provide a natural way to implement the abstract syntax in a compiler. To give an example, the following Haskell type defines the abstract syntax of the lambda calculus with abstractions, applications, and variables. For more information about using algebraic data types to represent abstract syntax for programming languages, see for example Appel's (1997) book on compiler construction in ML.

```
data Exp = EAbs String Exp | EApp Exp Exp | EVar String
```

Pattern matching is the technique for defining functions on algebraic data types. These functions are typically recursive. An example is a function that renames all the variables in an expression by prepending an underscore to their names:

```
rename :: Exp → Exp
rename e = case e of
             EAbs x b → EAbs ("_" ++ x) (rename b)
             EApp c a → EApp (rename c) (rename a)
             EVar x   → EVar ("_" ++ x)
```

3 Compositional Functions

Many functions used in compilers are *compositional*, in the sense that the result for a complex argument is constructed from the results for its parts. The *rename* function is an example of this. The essence of compositional functions is defined by the following higher-order function:

```
composOp :: (Exp → Exp) → Exp → Exp
composOp f e = case e of
                 EAbs x b → EAbs x (f b)
                 EApp c a → EApp (f c) (f a)
                 _        → e
```

Its power lies in that it can be used when defining other functions, to take care of cases that are just compositional. Such is the EApp case in *rename*, which we thus omit by writing:

```
rename :: Exp → Exp
rename e = case e of
             EAbs x b → EAbs ("_" ++ x) (rename b)
             EVar x   → EVar ("_" ++ x)
             _        → composOp rename e
```

In general, an abstract syntax has many more constructors, and this pattern saves much more work. For instance, in the implementation of GF (Ranta 2004), the Exp type has 30 constructors, and *composOp* is used in more than 20 functions, typically covering 90 % of all cases.

A major restriction of *composOp* is that its return type is Exp. How do we use it if we want to return something else? If we simply want to compute some result using the abstract syntax tree, without modifying the tree, we can use *composFold*:

```
composFold :: Monoid o ⇒ (Exp → o) → Exp → o
composFold f e = case e of
                   EAbs x b → f b
                   EApp c a → f c ⊕ f a
                   _        → ∅
```

This function takes an argument which maps terms to a monoid, and combines the results. The Monoid class requires an identity element $\emptyset$, which we return for leaf nodes, and an associative operation ($\oplus$), which we use to combine results from nodes with more than one child.

```
class Monoid a where
  ∅ :: a
  (⊕) :: a → a → a
```

Using *composFold* we can now, for example, write a function which gets the names of all free variables in an expression:

```
free :: Exp → Set String
free e = case e of
           EAbs x b → free b \ {x}
           EVar x   → {x}
           _        → composFold free e
```

This example uses a Set type with the operations $\setminus$ (set minus), $\{\cdot\}$ (singleton set), $\varnothing$ (empty set) and $\cup$ (union), with a Monoid instance such that $\emptyset = \varnothing$ and $(\oplus) = \cup$.

3.1 Monadic compositional functions

When defining a compiler in Haskell, it is convenient to use monads instead of plain functions, to deal with errors, state, etc. To this end, we generalize *composOp* to a monadic variant:

```
composM :: Monad m ⇒ (Exp → m Exp) → Exp → m Exp
composM f e = case e of
  EAbs x b → f b >>= (λb' → return (EAbs x b'))
  EApp c a → f c >>= (λc' → f a >>= (λa' → return (EApp c' a')))
  _        → return e
```

Here we are using the Monad type class (Peyton Jones 2003b):

```
class Monad m where
  (>>=)  :: m a → (a → m b) → m b
  return :: a → m a
```

If we want to maintain some state across the computation over the tree, we can use *composM* with a state monad (Jones 1995). In the example below, we will use a state monad State with these operations:

```
readState  :: State s s
writeState :: s → State s ()
runState   :: s → State s a → (a, s)
```

Now we can, for example, write a function that gives fresh names of the form `"_n"`, where n is an integer, to all bound variables in an expression. Here the state is an infinite supply of fresh variable names, and we pass a table of the new names for the bound variables to the recursive calls.

```
fresh :: Exp → Exp
fresh = fst ∘ runState names ∘ f []
  where names = ["_" ++ show n | n ← [0..]]
        f :: [(String, String)] → Exp → State [String] Exp
        f vs t = case t of
                   EAbs x b → do x' : ns ← readState
                                 writeState ns
                                 b' ← f ((x, x') : vs) b
                                 return (EAbs x' b')
                   EVar x   → return (EVar (findWithDefault x x vs))
                   _        → composM (f vs) t
findWithDefault :: Eq a ⇒ b → a → [(a, b)] → b
findWithDefault d _ [] = d
findWithDefault d k ((x, y) : xs) | x == k    = y
                                  | otherwise = findWithDefault d k xs
```

3.2 Generalizing *composOp*, *composM* and *composFold*

The three functions which we introduced above, henceforth referred to as *compositional operations*, share a common structure which we will now reveal. McBride and Paterson (2008) introduce *applicative functors*, which generalize monads. An applicative functor has two operations, *pure* and ⊛:

```
class Applicative f where
  pure :: a → f a
  (⊛)  :: f (a → b) → f a → f b
```

The *pure* operation corresponds to the *return* operation of a monad, and ⊛ corresponds to *ap*, which can be defined using ⪢=:

```
ap :: Monad m ⇒ m (a → b) → m a → m b
ap mf mx = mf ⪢= λf → mx ⪢= λx → return (f x)
```

We can rewrite *composM* to use *ap* instead of ⪢=:

```
composM :: Monad m ⇒ (Exp → m Exp) → Exp → m Exp
composM f e = case e of
                EAbs x b → return EAbs `ap` return x `ap` f b
                EApp c a → return EApp `ap` f c `ap` f a
                _        → return e
```

Since *composM* only uses *return* and *ap*, it actually works on all applicative functors, not just on monads. We call this generalized version *compos*:

```
compos :: Applicative f ⇒ (Exp → f Exp) → Exp → f Exp
compos f e = case e of
               EAbs x b → pure EAbs ⊛ pure x ⊛ f b
               EApp g h → pure EApp ⊛ f g ⊛ f h
               _        → pure e
```

By using wrapper types with appropriate Applicative instances (McBride and Paterson 2008), we can now define *composOp*, *composM* and *composFold* in terms of *compos*. The definitions of *composOp* and *composFold* are identical to McBride and Paterson's definitions of *fmap* and *accumulate* in terms of *traverse*, and the definition of *composM* follows directly from the relationship between applicative functors and monads.

```
composOp :: (Exp → Exp) → Exp → Exp
composOp f = runIdentity ∘ compos (Identity ∘ f)

newtype Identity a = Identity {runIdentity :: a}

instance Applicative Identity where
  pure                    = Identity
  Identity f ⊛ Identity x = Identity (f x)
```

```
composM :: Monad m ⇒ (Exp → m Exp) → Exp → m Exp
composM f = unwrapMonad ∘ compos (WrapMonad ∘ f)
newtype WrappedMonad m a = WrapMonad {unwrapMonad :: m a}
instance Monad m ⇒ Applicative (WrappedMonad m) where
  pure                              = WrapMonad ∘ return
  WrapMonad f ⊛ WrapMonad v = WrapMonad (f `ap` v)

composFold :: Monoid o ⇒ (Exp → o) → Exp → o
composFold f = getConst ∘ compos (Const ∘ f)
newtype Const a b = Const {getConst :: a}
instance Monoid m ⇒ Applicative (Const m) where
  pure _              = Const ∅
  Const f ⊛ Const v = Const (f ⊕ v)
```

Further compositional operations, such as *composM_* below can be defined by using other wrapper types.

```
composM_ :: Monad m ⇒ (Exp → m ()) → Exp → m ()
composM_ f = unwrapMonad_ ∘ composFold (WrapMonad_ ∘ f)
newtype WrappedMonad_ m = WrapMonad_ {unwrapMonad_ :: m ()}
instance Monad m ⇒ Monoid (WrappedMonad_ m) where
  ∅                                 = WrapMonad_ (return ())
  WrapMonad_ x ⊕ WrapMonad_ y = WrapMonad_ (x ≫ y)
```

4 Systems of Data Types

4.1 Several algebraic data types

For many languages, the abstract syntax is not just one data type, but many, which are often defined by mutual induction. An example is the following simple imperative language with statements, expressions, variables, and types. In this language, statements that return values (such as assignments and blocks that end with a return statement) can be used as expressions.

```
data Stm = SDecl Typ Var | SAss Var Exp | SBlock [Stm] | SReturn Exp
data Exp = EStm Stm | EAdd Exp Exp | EVar Var | EInt Int
data Var = V String
data Typ = TInt | TFloat
```

We now need one *compos* function for each recursive type, and some of the recursive calls must be made on terms which have types different from that which the function was called on. This can be solved by taking several functions as arguments, one for each type.

```
compossStm :: Applicative f ⇒
              (Stm → f Stm, Exp → f Exp, Var → f Var, Typ → f Typ)
               → Stm → f Stm
compossStm (fs, fe, fv, ft) s = case s of
                                  SDecl x y  → pure SDecl   ⊛ ft x ⊛ fv y
                                  SAss x y   → pure SAss    ⊛ fv x ⊛ fe y
                                  SBlock xs  → pure SBlock  ⊛ traverse fs xs
                                  SReturn x  → pure SReturn ⊛ fe x
compossExp :: Applicative f ⇒
              (Stm → f Stm, Exp → f Exp, Var → f Var, Typ → f Typ)
               → Exp → f Exp
compossExp (fs, fe, fv, ft) e = case e of
                                  EAdd x y  → pure EAdd   ⊛ fe x ⊛ fe y
                                  EStm x    → pure EStm   ⊛ fs x
                                  EVar x    → pure EVar   ⊛ fv x
```

Note that the Typ function is not actually required in *composExp*, but we include it here for the sake of uniformity. We would also need to implement *composOp*, *composM*, *composFold* etc. for each of the types. Even though these implementations would be identical for all type families, it is difficult to provide generic implementations of them without resorting to multi-parameter type classes and functional dependencies, since the type of the function tuple will depend on the type family. With these functions, we can define a renaming function more easily than without *composOp*:

```
renameStm :: Stm → Stm
renameStm t = composOpStm (renameStm, renameExp,
                           renameVar, renameTyp)
renameExp :: Exp → Exp
renameExp t = composOpExp ( renameStm, renameExp,
                           renameVar, renameTyp)
renameVar :: Var → Var
renameVar (V x) = V ("_" ++ x)
renameTyp :: Typ → Typ
renameTyp t = t
```

We now need up to one extra function per type (for non-recursive types we can get away with passing *id*). In a large system, this can result in significant overhead. For example, the abstract syntax used in the Glasgow Haskell Compiler contains more than 50 data types (Peyton Jones 2007).

4.2 Categories and trees

An alternative to separate mutual data types for abstract syntax is to define just one type Tree, whose constructors take Trees as arguments:

```
data Tree = SDecl Tree Tree | SAss Tree Tree | SBlock [Tree] | SReturn Tree
          | EStm Tree | EAdd Tree Tree | EVar Tree | EInt Int
          | V String | TInt | TFloat
```

This is essentially the representation one would use in a dynamically typed language. It does not, however, constrain the combinations enough for our liking: there are many Trees that are even syntactically nonsense.

A solution to this problem is provided by dependent types (Martin-Löf 1984; Nordström et al. 1990). Instead of a constant type Tree, we define an **inductive family** (Dybjer 1994) Tree c, indexed by a **category** c. The category is just a label to distinguish between different types of trees. Inductive families have previously been used for representing the abstract syntax of **well-typed expressions**: the family Exp a gives separate, yet related, types to integer expressions, boolean expressions, etc. (Augustsson and Petersson 1994). The extension from such a family to one comprising all syntactic categories (expressions, statements, etc.) seems to be a novelty of our work. We must now leave standard Haskell and use a Haskell-like language with dependent types and inductive families. Agda (Coquand and Coquand 1999; Norell 2007) is one such language. What one would define in Agda is an enumerated type:

```
data Cat = Stm | Exp | Var | Typ
```

followed by an **idata** (inductive data type, or in this case an inductive family of data types) definition of Tree, indexed on Cat. We omit the Agda definitions of the Tree family and the *compos* function as they are virtually identical to the Haskell versions shown below, except that in Agda the index for Tree is a value of type Cat, whereas in Haskell the index is a dummy data type.

We can also do our exercise with a limited form of dependent types provided by Haskell since GHC 6.4: **Generalized Algebraic Data Types** (GADTs, Peyton Jones et al. 2006; Augustsson and Petersson 1994). We cannot quite define a *type* of categories, but we can define a set of dummy data types:

```
data Stm
data Exp
data Var
data Typ
```

To define the inductive family of trees, we write, in this extension of Haskell:

```
data Tree :: * → * where
  SDecl   :: Tree Typ → Tree Var → Tree Stm
  SAss    :: Tree Var → Tree Exp → Tree Stm
  SBlock  :: [Tree Stm] → Tree Stm
  SReturn :: Tree Exp → Tree Stm
  EStm    :: Tree Stm → Tree Exp
  EAdd    :: Tree Exp → Tree Exp → Tree Exp
  EVar    :: Tree Var → Tree Exp
```

```
EInt    :: Int → Tree Exp
V       :: String → Tree Var
TInt    :: Tree Typ
TFloat  :: Tree Typ
```

In Haskell we cannot restrict the types used as indices in the Tree family, which makes it entirely possible to construct types such as Tree String. However, since there are no constructors targeting this type, $\bot$ is the only element in it.

4.3 Compositional operations

The power of inductive families is shown in the definition of the function *compos*. We now define it simultaneously for the whole syntax, and can then use it to define tree-traversing programs concisely.

```
compos :: Applicative f ⇒ (∀a. Tree a → f (Tree a)) → Tree c → f (Tree c)
compos f t = case t of
               SDecl x y  → pure SDecl  ⊛ f x ⊛ f y
               SAss x y   → pure SAss   ⊛ f x ⊛ f y
               SBlock xs  → pure SBlock ⊛ traverse f xs
               SReturn x  → pure SReturn ⊛ f x
               EAdd x y   → pure EAdd   ⊛ f x ⊛ f y
               EStm x     → pure EStm   ⊛ f x
               EVar x     → pure EVar   ⊛ f x
               _          → pure t
```

The first argument must now be polymorphic, since it is applied to subtrees of different types. This requires rank-2 polymorphism (Leivant 1983; Peyton Jones et al. 2007), a widely supported Haskell extension. The argument to the SBlock constructor is a list of statements, which we handle by visiting the list elements from left to right, using the *traverse* function (McBride and Paterson 2008), which generalizes *mapM*:

```
traverse :: Applicative f ⇒ (a → f b) → [a] → f [b]
traverse f []       = pure []
traverse f (x : xs) = pure (:) ⊛ f x ⊛ traverse f xs
```

The other compositional operations are special cases of *compos* in the same way as before.

4.4 A library of compositional operations

Since all the other compositional operations can be defined in terms of *compos*, we create a type class containing the *compos* function, and define the other operations in terms of it. The code for this is shown in Figure 1.

```
class Compos t where
  compos :: Applicative f ⇒ (∀a. t a → f (t a)) → t c → f (t c)
composOp :: Compos t ⇒ (∀a. t a → t a) → t c → t c
composOp f = runIdentity ∘ compos (Identity ∘ f)
composFold :: (Monoid o, Compos t) ⇒ (∀a. t a → o) → t c → o
composFold f = getConst ∘ compos (Const ∘ f)
composM :: (Compos t, Monad m) ⇒ (∀a. t a → m (t a)) → t c → m (t c)
composM f = unwrapMonad ∘ compos (WrapMonad ∘ f)
composM_ :: (Compos t, Monad m) ⇒ (∀a. t a → m ()) → t c → m ()
composM_ f = unwrapMonad_ ∘ composFold (WrapMonad_ ∘ f)
```

Figure 1. The Compos module.

4.5 Migrating existing programs

Replacing a family of data types with a GADT (Generalized Algebraic Data Type) does not change the appearance of the expressions and patterns in the syntax tree types. However, the types now have the form Tree c. If we want, we can give the dummy types names other than those of the original categories, for example Stm_, Exp_, Var_, and Typ_, and use type synonyms to make the types also look like they did when we had multiple data types:

```
type Stm = Tree Stm_
type Exp = Tree Exp_
type Var = Tree Var_
type Typ = Tree Typ_
```

This allows us to modify existing programs to switch from a family of data types to a GADT, simply by replacing the type definitions. All existing functions remain valid with the new type definitions, which makes it possible to take advantage of our compositional operations when writing new functions, without being forced to change any existing ones. There are a few minor issues: the limitations on type inference for GADTs (Peyton Jones et al. 2006) and rank-2 polymorphism (Peyton Jones et al. 2007) may require type signatures for some functions, and since GHC does not currently support type class instance deriving for GADTs, we have to write instances of common type classes such as Show and Eq for our type family by hand.

4.6 Examples

Example: Variable renaming

It is laborious to define a renaming function for the original Haskell definition with separate data types (as shown in Section 4.1). But now it is easy:

```
rename :: Tree c → Tree c
rename t = case t of
             V x → V ("_" ++ x)
             _   → composOp rename t
```

Example: Symbol table construction

This function constructs a variable symbol table by folding over the syntax tree. We use the Monoid instance for lists, where the associative operation is ++, and the identity element is [].

```
symbols :: Tree c → [(Tree Var, Tree Typ)]
symbols t = case t of
              SDecl typ var → [(var, typ)]
              _             → composFold symbols t
```

Example: Syntactic sugar

This example shows how easy it is to add syntax constructs as syntactic sugar, i.e. syntactic constructs that can be eliminated. Suppose that you want to add increment statements. This means a new branch in the definition of Tree c from Section 4.2:

```
SIncr :: Tree Var → Tree Stm
```

Increments are eliminated by translation to assignments as follows:

```
elimIncr :: Tree c → Tree c
elimIncr t = case t of
               SIncr v → SAss v (EAdd (EVar v) (EInt 1))
               _       → composOp elimIncr t
```

Example: Warnings for assignments

To encourage pure functionality, this function sounds the bell each time an assignment occurs. Since we are not interested in the return value of the function, but only in its IO outputs, we use the function *composM_* (like *composM* but without a tree result, see Section 3.2 for its definition).

```
warnAssign :: Tree c → IO ()
warnAssign t = case t of
                 SAss _ _ → putChar (chr 7)
                 _        → composM_ warnAssign t
```

Example: Constant folding

We want to replace additions of constants by their result. Here is a first attempt:

```
constFold :: Tree c → Tree c
constFold e = case e of
                EAdd (EInt x) (EInt y) → EInt (x + y)
                _                      → composOp constFold e
```

This works for simple cases, but what about for example $1 + (2 + 3)$? This is an addition of constants, but is not matched by our pattern above. We have to look at the results of the recursive calls:

```
constFold' :: Tree c → Tree c
constFold' e = case e of
                 EAdd x y → case (constFold' x, constFold' y) of
                              (EInt n, EInt m) → EInt (n + m)
                              (x', y')         → EAdd x' y'
                 _        → composOp constFold' e
```

This illustrates a common pattern used when the recursive calls can introduce terms which we want to handle.

4.7 Writing Compos instances

Up til now, we have only shown *compos* functions for example data types. But what is the general pattern? We will consider types of the form:

```
data T : ∗ → ∗ where
  C1 :: t1,1 → ... → t1,a1 → T c1
  ...
  Cn :: tn,1 → ... → tn,an → T cn
```

where $n \geqslant 0$ is the number of data constructors, $a_x \geqslant 0$ is the arity of data constructor C_x, $t_{x,y}$ is the type of argument y of constructor C_x, and c_x is the type argument to T in the type of constructor C_x. The argument types $t_{x,y}$ cannot be type variables, since it must be possible to determine statically whether or not each argument belongs to the type family T. All Compos instances have this general form:

```
instance Compos T where
  compos f t = case t of
                 C1 b1 ... ba1 → pure C1 ⊛ g1,1 b1 ⊛ ... ⊛ g1,a1 ba1
                 ...
                 Cn b1 ... ban → pure Cn ⊛ gn,1 b1 ⊛ ... ⊛ gn,an ban
```

where each $g_{x,y}$ function depends on the type $t_{x,y}$ of the corresponding constructor argument. There is some freedom in how $g_{x,y}$ is chosen. The simplest

choice is to only use f on children which have a type in the type family T:

$$g_{x,y} = \begin{cases} f & \text{if } \exists c.t_{x,y} = \mathsf{T}\ c \\ \mathit{pure} & \text{otherwise} \end{cases}$$

In the *compos* implementation shown in Section 4.3, we used *traverse* to map f over any lists containing elements in the type family T. This can be generalized to any traversable type, using the Traversable type class by McBride and Paterson (2008).

$$g_{x,y} = \begin{cases} f & \text{if } \exists c.t_{x,y} = \mathsf{T}\ c \\ \mathit{traverse}\ f & \text{if } \exists c.t_{x,y} = \mathsf{F}\ (\mathsf{T}\ c) \wedge \mathsf{Traversable\ F} \\ \mathit{pure} & \text{otherwise} \end{cases}$$

Parameterized abstract syntax

We may want to have type parameters for the entire type family. For example, GHC's abstract syntax is parameterized over the type of identifiers. This makes it possible to use the same abstract syntax, with different identifier types, for the input before and after name resolution. We can add extra type parameters to our type family to support this. For example:

```
data Decl
data Exp

data Tree :: * → * → * where
  Decl :: i → Tree i Exp → Tree i Decl
  App  :: Tree i Exp → Tree i Exp → Tree i Exp
  Var  :: i → Tree i Exp
```

We said above that constructors should not have type variable arguments, but when we implement *compos*, we can choose to treat i as a non-Tree type.

An optimization

As done in the *compos* implementations in Sections 3.2 and 4.3, the cases for all non-recursive constructors (i.e. constructors C_x such that $\forall y.g_{x,y} = \mathit{pure}$) can be optimized to a single catch-all case: $\_ \to \mathit{pure}\ t$. This can be done since $\mathit{pure}\ \mathsf{C}_x \circledast \mathit{pure}\ b_1 \circledast \ldots \circledast \mathit{pure}\ b_{a_x} = \mathit{pure}\ (\mathsf{C}_x\ b_1 \ldots b_{a_x})$ by the homomorphism law for applicative functors (see Section 4.8).

4.8 Properties of compositional operations

The following laws hold for our compositional operations:

Identity 1	$\mathit{compos}\ \mathit{pure} = \mathit{pure}$
Identity 2	$\mathit{composOp}\ \mathit{id} = \mathit{id}$
Identity 3	$\mathit{composFold}\ (\lambda\_ \to \emptyset) = \lambda\_ \to \emptyset$
Composition	$\mathit{composOp}\ f \circ \mathit{composOp}\ g = \mathit{composOp}\ (f \circ g)$

Here, $=$ denotes extensional function equality at some type T for which we have defined *compos* according to the scheme shown in Section 4.7. That is, $f = g$ means that for all total values $t :: \mathsf{T}$, $f\ t = g\ t$. In the proofs, we will make use of the laws for applicative functors (McBride and Paterson 2008):

Identity	$\mathit{pure}\ \mathit{id} \circledast u = u$
Composition	$\mathit{pure}\ (\circ) \circledast u \circledast v \circledast w = u \circledast (v \circledast w)$
Homomorphism	$\mathit{pure}\ f \circledast \mathit{pure}\ x = \mathit{pure}\ (f\ x)$
Interchange	$u \circledast \mathit{pure}\ x = \mathit{pure}\ (\lambda f \to f\ x) \circledast u$

We would like *compos* to have the property that it does not modify the term on its own, i.e. that:

Theorem 1. *For all total values $t :: \mathsf{T}$, compos pure t = pure t.*

Proof. Consider some $t = \mathsf{C}\ t_1 \ldots t_n$, where C is an arbitrary constructor of T with arity n. The relevant part of the *compos* function is then:

$$\begin{aligned}&\mathit{compos}\ f\ t = \mathbf{case}\ t\ \mathbf{of}\\ &\qquad\qquad \mathsf{C}\ x_1 \ldots x_n \to \mathit{pure}\ \mathsf{C} \circledast g_1\ x_1 \circledast \ldots \circledast g_n\ x_n\end{aligned}$$

where each g_i is either *pure*, f, or *traverse* f, depending on the type of x_i. Since $f = \mathit{pure}$ in the case that we are reasoning about, the functions $g_1 \ldots g_n$ are either *pure* or *traverse pure*. As noted by Gibbons and Oliveira (2006), all implementations of *traverse* should satisfy the "purity law" *traverse pure* = *pure*. Thus, all the $g_1 \ldots g_n$ functions are *pure* and the constructor cases all have the form:

$$\mathsf{C}\ x_1 \ldots x_n \to \mathit{pure}\ \mathsf{C} \circledast \mathit{pure}\ x_1 \circledast \ldots \circledast \mathit{pure}\ x_n$$

By repeated use of the homomorphism law for applicative functors, we have that:

$$\mathit{pure}\ \mathsf{C} \circledast \mathit{pure}\ x_1 \circledast \ldots \circledast \mathit{pure}\ x_n = \mathit{pure}\ (\mathsf{C}\ x_1 \ldots x_n)$$

Thus, for all total $t : \mathsf{T}$, *compos pure* t = *pure* t. □

With the definitions of *composOp* and *composFold* given in Section 4.3, **Identity 2** and **Identity 3** follow straightforwardly from Theorem 1.

Theorem 2. *For all total $t :: \mathsf{T}$, composOp f (composOp g t) = composOp $(f \circ g)$ t.*

Proof. Consider some $t = \mathsf{C}\ t_1 \ldots t_n$, where C is an arbitrary constructor of T, with arity n. As in the proof of Theorem 1, the interesting part of *compos* is:

$$\begin{aligned}&\mathit{compos}\ f\ t = \mathbf{case}\ t\ \mathbf{of}\\ &\qquad\qquad \mathsf{C}\ x_1 \ldots x_n \to \mathit{pure}\ \mathsf{C} \circledast g_1\ x_1 \circledast \ldots \circledast g_n\ x_n\end{aligned}$$

Lemma 1. *composOp g* ($\mathsf{C}\ x_1 \ldots x_n$) = $\mathsf{C}\ (g'_1\ x_1) \ldots (g'_n\ x_n)$*, where each* g'_i *is id, g or fmap g, depending on the type of* x_i*.*

Proof. *composOp g* ($\mathsf{C}\ x_1 \ldots x_n$)
= { Definition of *composOp* }
runIdentity (*compos* (Identity $\circ$ *g*) ($\mathsf{C}\ x_1 \ldots x_n$))
= { Definition of *compos* }
runIdentity (*pure* $\mathsf{C} \circledast g_1\ x_1 \circledast \ldots \circledast g_n\ x_n$)
= { Definition of *pure* for Identity }
runIdentity (Identity $\mathsf{C} \circledast g_1\ x_1 \circledast \ldots \circledast g_n\ x_n$)
= { Definition of $\circledast$ for Identity }
runIdentity (Identity (C (*runIdentity* ($g_1\ x_1$))) $\circledast \ldots \circledast g_n\ x_n$)
= { Definition of $\circledast$ for Identity }
runIdentity (Identity (C (*runIdentity* ($g_1\ x_1$)) $\ldots$ (*runIdentity* ($g_n\ x_n$))))
= { Introduce g'_i = *runIdentity* $\circ\ g_i$ }
runIdentity (Identity (C ($g'_1\ x_1$) $\ldots$ ($g'_n\ x_n$)))
= { Definition of *runIdentity* }
$\mathsf{C}\ (g'_1\ x_1) \ldots (g'_n\ x_n)$

Since each g_i is Identity, Identity $\circ\ g$ or *traverse* (Identity $\circ\ g$), each g'_i is *id*, *g* or *fmap g*. The last case relies on the observation by Gibbons and Oliveira (2006) that all implementations of *traverse* should satisfy *traverse* (Identity $\circ\ f$) = Identity $\circ$ *fmap f*. □

Now,

composOp f (*composOp g* ($\mathsf{C}\ x_1 \ldots x_n$))
= { Lemma 1 }
composOp f ($\mathsf{C}\ (g'_1\ x_1) \ldots (g'_n\ x_n)$)
= { Lemma 1 }
$\mathsf{C}\ (f'_1\ (g'_1\ x_1)) \ldots (f'_n\ (g'_n\ x_n))$
= { Definition of $\circ$ }
$\mathsf{C}\ ((f'_1 \circ g'_1)\ x_1) \ldots ((f'_n \circ g'_n)\ x_n)$
= { Lemma 1 and *fmap f* $\circ$ *fmap g* = *fmap* ($f \circ g$) }
composOp ($f \circ g$) ($\mathsf{C}\ x_1 \ldots x_n$)

□

One may think that the stronger *compos g t* $\gg\!=$ *compos f* = *compos* ($\lambda x \rightarrow g\ x \gg\!= f$) *t* would hold for any Applicative type that is also a Monad, but it does not, as it changes the order of the monadic computations.

It should also be possible to perform formal reasoning about our compositional operations using dependent type theory with tree sets, as discussed in Section 7.4.

5 Almost Compositional Functions and the Visitor Design Pattern

The Visitor design pattern (Gamma et al. 1995) is a pattern used in object-oriented programming to define an operation for each of the concrete elements of an object hierarchy. We will show how an adaptation of the Visitor pattern can be used to define almost compositional functions in object-oriented languages, in a manner quite similar to that shown above for languages with algebraic data types and pattern matching.

First we present the object hierarchies corresponding to the algebraic data types. Each object hierarchy has a generic Visitor interface. We then show a concrete visitor that corresponds to the *composOp* function. Our examples are written in Java 1.5 (Gosling et al. 2005) and make use of its parametric polymorphism (Bracha et al. 1998).

5.1 Abstract syntax representation

We use a standard encoding of abstract syntax trees in Java (Appel 2002), along with the support code for a type-parametrized version of the Visitor design pattern. For each algebraic data type in the Haskell version (as shown in Section 4.1), we have an abstract base class in the Java representation:

```
public abstract class Stm {
  public abstract⟨R, A⟩R accept (Visitor⟨R, A⟩v, A arg);
  public interface Visitor⟨R, A⟩ {
    public R visit (SDecl p, A arg);
    public R visit (SAss p, A arg);
    public R visit (SBlock p, A arg);
    public R visit (SReturn p, A arg);
    public R visit (SInc p, A arg);
  }
}
```

The base class contains an interface for visitors, with methods for visiting each of the inheriting classes. The Visitor interface has two type parameters: R is the type of the value returned by the Visitor, and A is the type of an auxiliary argument which is threaded through the traversal. Each inheriting class must have a method for accepting the visitor. This method dispatches the call to the correct method in the visitor.

For each data constructor in the algebraic data type, we have a concrete class which inherits from the abstract base class, for example:

```
public class SDecl extends Stm {
  public final Typ typ_;
  public final Var var_;
  public SDecl (Typ p1, Var p2) {typ_ = p1; var_ = p2; }
```

```
    public⟨R, A⟩R accept (Visitor⟨R, A⟩v, A arg) {
      return v.visit (this, arg);
    }
}
```

The Visitor interface can be used to define operations on all the concrete classes in one or more of the hierarchies (when defining an operation on more than one hierarchy, the visitor implements multiple Visitor interfaces). This corresponds to the initial examples of pattern matching on all of the constructors, as shown in Section 2. It suffers from the same problem: lots of repetitive traversal code.

5.2 ComposVisitor

We can create a class which does all of the traversal and tree rebuilding. This corresponds to the *composOp* function in the Haskell implementation.

```
public class ComposVisitor⟨A⟩implements
  Stm.Visitor⟨Stm, A⟩, Exp.Visitor⟨Exp, A⟩,
  Var.Visitor⟨Var, A⟩, Typ.Visitor⟨Typ, A⟩ {
  public Stm visit (SDecl p, A arg) {
    Typ typ_ = p.typ_.accept (this, arg);
    Var var_ = p.var_.accept (this, arg);
    return new SDecl (typ_, var_);
  }
  //...
}
```

The ComposVisitor class implements all the Visitor interfaces in the abstract syntax, and can thus visit all of the constructors in all of the types. Each *visit* method visits the children of the current node, and then constructs a new node with the results returned from these visits. A visitor for a given base class corresponds to a Haskell case expression on an algebraic data type. Multiple interface inheritance lets us write a single visitor which can handle multiple classes. Such a visitor is then like a case expression on an entire type family. This use of multiple interface inheritance is what makes it possible to handle the multiple-type recursion issue that forced us to use GADTs and rank-2 polymorphism in Haskell.

The code above could be optimized to eliminate the reconstruction overhead when the recursive calls do not modify the subtrees. For example, if all the objects which are being traversed are immutable, unnecessary copying could be avoided by doing a pointer comparison between the old and the new child. If all the children are unchanged, we do not need to construct a new parent.

5.3 Using ComposVisitor

While the *composOp* function takes a function as a parameter, and applies that function to each constructor argument, the ComposVisitor class in itself is essentially a complicated implementation of the identity function. Its power comes from the fact that we can override individual *visit* methods.

When using the standard Visitor pattern, adding new operations is easy, but adding new elements to the object hierarchy is difficult, since it requires changing the code for all the operations. Having a ComposVisitor changes this, as we can add a new element, and only have to change the Visitor interface, the ComposVisitor, and any operations which need to have special behavior for the new class.

The Java code below implements the desugaring example from Section 4.6 where increments are replaced by addition and assignment. Note that in Java we only need the interesting case, all the other cases are taken care of by the parent class.

```
class Desugar extends ComposVisitor⟨Object⟩ {
  public Stm visit (SInc i, Object arg) {
    Exp rhs = new EAdd (new EVar (i.var_), new EInt (1));
    return new SAss (i.var_, rhs);
  }
}
Stm desugar (Stm stm) {
  return stm.accept (new Desugar (), null);
}
```

The Object argument to the *visit* method is a dummy since this visitor does not need any extra arguments. The *desugar* method at the end is just a wrapper used to hide the details of getting the visitor to visit the statement, and passing in the dummy argument.

This being an imperative language, we do not have to do anything special to thread a state through the computation. Here is the symbol table construction function from Section 4.6 in Java:

```
class BuildSymTab extends ComposVisitor⟨Object⟩ {
  Map⟨Var, Typ⟩ symTab = new HashMap⟨Var, Typ⟩();
  public Stm visit (SDecl d, Object arg) {
    symTab.put (d.var_, d.typ_);
    return d;
  }
}
Map⟨Var, Typ⟩ symbolTable (Stm stm) {
  BuildSymTab v = new BuildSymTab ();
  stm.accept (v, null);
  return v.symTab;
}
```

You may wonder why this function was implemented as a stateful computation instead of as a fold like in the Haskell version. Creating a visitor which corresponds to *composFold* would be less elegant in Java, since we would have to pass a combining function and a base case value to the visitor. This could be done by adding abstract methods in the visitor, but in most cases the stateful implementation is probably more idiomatic in Java.

Our final Java example is the example from Section 3, where we compute the set of free variables in a term in the small functional language introduced in Section 2.

```
class Free extends ComposVisitor⟨Set⟨String⟩⟩ {
  public Exp visit (EAbs e, Set⟨String⟩ vs) {
    Set⟨String⟩ xs = new TreeSet⟨String⟩();
    e.exp_.accept (this, xs);
    xs.remove (e.ident_);
    vs.addAll (xs);
    return e;
  }
  public Exp visit (EVar e, Set⟨String⟩ vs) {
    vs.add (e.ident_);
    return e;
  }
}
Set⟨String⟩ freeVars (Exp exp) {
  Set⟨String⟩ vs = new TreeSet⟨String⟩();
  exp.accept (new Free (), vs);
  return vs;
}
```

Here we make use of the possibility of passing an extra argument to the *visit* methods. The argument is a set to which the *visit* method adds all the free variables in the visited term.

6 Language and Tool Support for Compositional Operations

When using the method we have described, one needs to define the Haskell Compos instance or Java ComposVisitor class manually for each type or type family. To create Compos instances automatically, we could extend the Haskell compiler to allow deriving instances of Compos. Another possibility would be to generate the instances using Template Haskell (Sheard and Peyton Jones 2002), DrIFT (Winstanley et al. 2007), or Derive (Mitchell and O'Rear 2007), but these tools do not yet support GADTs.

We have added a new back-end to the BNF Converter (BNFC) (Forsberg 2007; Forsberg and Ranta 2006) tool which generates a Haskell GADT abstract

```
SDecl.  Stm ::= Typ Var ";";
SAss.   Stm ::= Var "=" Exp ";";
SBlock. Stm ::= "{" [Stm] "}";
SReturn.Stm ::= "return" Exp ";";
SInc.   Stm ::= Var "++" ";";
separator Stm "";

EStm.   Exp1 ::= Stm;
EAdd.   Exp1 ::= Exp1 "+" Exp2;
EVar.   Exp2 ::= Var;
EInt.   Exp2 ::= Integer;
EDbl.   Exp2 ::= Double;
coercions Exp 2;

V.      Var  ::= Ident;

TInt.   Typ ::= "int";
TDbl.   Typ ::= "double";
```

Figure 2. LBNF grammar for the simple imperative language.

syntax type along with instances of Compos, Eq, Ord and Show. We have also extended the BNFC Java 1.5 back-end to generate the Java abstract syntax representation shown above, along with the ComposVisitor class. In addition to the abstract syntax types and traversal components described in this paper, the generated code also includes a lexer, a parser, and a pretty printer. We can generate all the Haskell or Java code for our simple imperative language example using the grammar shown in Figure 2. It is written in LBNF (Labelled Backus-Naur Form), the input language for BNFC.

7 Related Work

7.1 Scrap Your Boilerplate

The part of this work dealing with functional programming languages can be seen as a solution to a subset of the problems solved by generic programming systems. Like "Scrap Your Boilerplate" (SYB) (Lämmel and Peyton Jones 2003), we focus on traversal operations that make it easier to write functions over a given rich data type or set of data types when there are only a few "interesting" cases. Our approach does not aim at defining functions such as equality, hashing, or pretty-printing, which need to consider every constructor in the type or type family. We also do not address the problem of writing *polytypic functions* (Jansson and Jeuring 1997; Hinze 2004), that is, functions that work on any data type, even those which are yet to be defined.

Introduction to Scrap Your Boilerplate

SYB uses generic traversal functions along with a type safe cast operation implemented by the use of type classes. This allows the programmer to extend fully generic operations with type-specific cases, and use these with various traversal schemes. Data types must have instances of the Typeable and Data type classes to be used with SYB.

The original "Scrap Your Boilerplate" paper (Lämmel and Peyton Jones 2003) contains a number of examples, some of which we will show as an introduction and later use for comparison. In the examples, some type synonyms (GenericT and GenericQ) have been inlined to make the function types more transparent. The examples work on a family of data types:

```
data Company = C [Dept]                  deriving (Typeable, Data)
data Dept    = D Name Manager [Unit]     deriving (Typeable, Data)
data Unit    = PU Employee | DU Dept     deriving (Typeable, Data)
data Employee = E Person Salary          deriving (Typeable, Data)
data Person  = P Name Address            deriving (Typeable, Data)
data Salary  = S Float                   deriving (Typeable, Data)
type Manager = Employee
type Name    = String
type Address = String
```

The first example increases the salary of all employees:

```
increase :: Data a => Float -> a -> a
increase k = everywhere (mkT (incS k))

incS :: Float -> Salary -> Salary
incS k (S s) = S (s * (1 + k))
```

The *everywhere* function applies a generic transformation to every node, bottom-up, and *mkT* makes a type specific transformation generic. More advanced traversal schemes are also supported. This example increases the salary of everyone in a named department:

```
incrOne :: Data a => Name -> Float -> a -> a
incrOne n k a | isDept n a = increase k a
              | otherwise  = gmapT (incrOne n k) a

isDept :: Data a => Name -> a -> Bool
isDept n = False `mkQ` isDeptD n

isDeptD :: Name -> Dept -> Bool
isDeptD n (D n' _ _) = n == n'
```

The *gmapT* function applies a generic transformation to the immediate subterms. SYB also supports queries, that is, functions that compute some result from the data structure rather than returning a modified structure. A type-specific query is made generic by *mkQ*, whose first argument is a constant that

is returned for all other types. This example computes the sum of the salaries of everyone in the company:

```
salaryBill :: Company → Float
salaryBill = everything (+) (0 `mkQ` billS)

billS :: Salary → Float
billS (S f) = f
```

The *everything* function applies a generic query everywhere in a term, and summarizes the results using the function given as the first argument.

SYB examples using compositional operations

We will now show the above examples implemented using our compositional operations. We lift the family of data types from the previous section into a GADT:

```
data Company; data Dept;   data Unit
data Employee; data Person; data Salary

type Manager = Employee
type Name    = String
type Address = String

data Tree :: ∗ → ∗ where
  C  :: [Tree Dept] → Tree Company
  D  :: Name → Tree Manager → [Tree Unit] → Tree Dept
  PU :: Tree Employee → Tree Unit
  DU :: Tree Dept → Tree Unit
  E  :: Tree Person → Tree Salary → Tree Employee
  P  :: Name → Address → Tree Person
  S  :: Float → Tree Salary
```

We define *compos* as described in Section 4.7, and use the operations from the library of compositional operations from Section 4.4 to implement the examples.

```
increase :: Float → Tree c → Tree c
increase k c = case c of
                 S s → S (s ∗ (1 + k))
                 _   → composOp (increase k) c
```

Here is the richer traversal example:

```
incrOne :: Name → Float → Tree c → Tree c
incrOne d k c = case c of
                  D n _ _ | n == d → increase k c
                  _                → composOp (incrOne d k) c
```

Query functions are also easy to implement (given a Monoid instance where $\emptyset = 0$ and $(\oplus) = (+)$):

```
salaryBill :: Tree c → Float
salaryBill c = case c of
                 S s → s
                 _   → composFold salaryBill c
```

These examples can all be written as single functions, whereas with SYB they each consist of two or three functions. SYB requires at least one function for each type-specific case, and one function that extends a generic traversal with the type specific cases.

SYB is a powerful system, but for many common uses such as the examples presented here, we believe that the *composOp* approach is more intuitive and easy to use. The drawback is that the data type family has to be lifted to a GADT, and that the *compos* function must be implemented. However, this only needs to be done once, and at least the latter can be automated, either by using BNFC, or by extending the Haskell compiler to generate instances of Compos (as is done for the Data and Typeable classes used by SYB).

Using SYB to implement compositional operations

Single data type Above we have shown how to replace simple uses of SYB with compositional operations. We will now show the opposite, and investigate to what extent the compositional operations can be reimplemented using SYB. The renaming example for the simple functional language, as shown in Section 3, looks very similar when implemented using SYB:

```
rename :: Exp → Exp
rename e = case e of
             EAbs x b → EAbs ("_" ++ x) (rename b)
             EVar x   → EVar ("_" ++ x)
             _        → gmapT (mkT rename) e
```

For the single data type case, our *composOp* and *composM* can be implemented with *gmapT* and *gmapM* (a monadic version of *gmapT*). The *gmapQ* function, which returns a list of the results of applying a query to the immediate subterms, can be used to write *composFold*. Our *compos* function can be written in terms of *gfoldl*, the one SYB function which can be used to implement all the others. Here are their definitions for the Exp type:

```
composOp :: (Exp → Exp) → Exp → Exp
composOp f = gmapT (mkT f)

composM :: Monad m ⇒ (Exp → m Exp) → Exp → m Exp
composM f = gmapM (mkM f)

composFold :: Monoid o ⇒ (Exp → o) → Exp → o
composFold f = foldl (⊕) ∅ ∘ gmapQ (mkQ ∅ f)

compos :: Applicative f ⇒ (Exp → f Exp) → Exp → f Exp
compos f = gfoldl (λx y → x ⊛ extM pure f y) pure
```

Here the *extM* function, which adds a type-specific case to a generic transformation, has been generalized to arbitrary functors (the *extM* from SYB requires a Monad).

Families of data types For the multiple data type case, it is difficult to use SYB to implement our examples with the desired type. When using *composOp*, the type restriction is achieved as a byproduct of lifting the family of data types into a GADT. Using a GADT to restrict the function types when using SYB is currently not practical, since current GHC versions cannot derive Data and Typeable instances automatically for GADTs. We can implement functions with types that are too general or too specific. For example, this is too general:

```
rename :: Data a ⇒ a → a
rename = gmapT (rename `extT` renameVar)
  where renameVar :: Var → Var
        renameVar (V x) = V ("_" ++ x)
renameStm :: Stm → Stm
renameStm = rename
```

What we would like to have is a *rename* function which can be applied to any abstract syntax tree, but not to things that are not abstract syntax trees. With a family of normal Haskell data types, the restriction could be achieved by the use of a dummy type class:

```
class Data a ⇒ Tree a
instance Tree Stm
instance Tree Exp
instance Tree Var
instance Tree Typ
renameTree :: Tree a ⇒ a → a
renameTree = rename
```

However, we would like the class Tree to be closed, something which is currently only achievable using hacks such as not exporting the class.

Using compositional operations to implement SYB

We can also try to implement the SYB functions in terms of our functions. If we are only interested in our single data type, this works:

```
gmapT :: Data a ⇒ (∀b. Data b ⇒ b → b) → a → a
gmapT f = mkT (composOp f)
gmapM :: (Data a, Monad m) ⇒ (∀b. Data b ⇒ b → m b) → a → m a
gmapM f = mkM (composM f)
gmapQ :: Data a ⇒ (∀b. Data b ⇒ b → u) → a → [u]
gmapQ f = mkQ [] (composFold (λx → [f x]))
```

Note that these functions are no longer truly generic: even though their types are the same as the SYB versions', they will only apply the function that they are given to values in the single data type Exp. Defining *gfoldl* turns out to be problematic, since the combining operation that *gfoldl* requires cannot be constructed from the operations of an applicative functor.

For the type family case, it does not seem possible to use compositional operations to implement SYB operations. It is even unclear what this would mean, since type families are implemented in different ways in the two approaches.

The Spine data type

In *"Scrap Your Boilerplate" Reloaded* (Hinze et al. 2006), SYB is explained by using a GADT called Type to lift all types into a single type Spine. For our type family example, this becomes:

```
data Stm; data Exp; data Var; data Typ
data Type :: ∗ → ∗ where
  Stm    :: Type Stm
  Exp    :: Type Exp
  Var    :: Type Var
  Typ    :: Type Typ
  List   :: Type a → Type [a]
  Int    :: Type Int
  String :: Type String
data Typed a = a : Type a
data Spine :: ∗ → ∗ where
  Constr :: a → Spine a
  (⋄)    :: Spine (a → b) → Typed a → Spine b
```

For example, the value EVar (V "x") is represented as Constr EVar ⋄ V "x" : Var. Compared to our representation, the Spine data type only lifts the top-level (or spine) of the value, rather than the entire value. The Spine type adds another level above the existing types, instead of replacing them, which changes how values are written. It also decouples constructors from their arguments, making it impossbile to do pattern matching directly. While this means that the Spine type cannot be used to replace our type family representation, it can be used to implement the SYB combinators. Thus it can be used to implement *compos* as is shown in Section 7.1.

Scrap Your Boilerplate conclusions

We consider the main differences between Scrap Your Boilerplate and our compositional operations to be that:

- When using SYB, no changes to the data types are required (except some type class deriving), but the way in which functions over the data types

are written is changed drastically. With compositional operations on the other hand, the data type family must be lifted to a GADT, while the style in which functions are written remains more natural.

- SYB functions over multiple data types are too generic, in that they are not restricted to the type family for which they are intended.
- Our approach is a general pattern which can be translated rather directly to other programming languages and paradigms.
- Compositional operations directly abstract out the pattern matching, recursion and reconstruction code otherwise written by hand. SYB uses runtime type representations and type casts, which gives more genericity, at the expense of transparency and understandability.

7.2 Catamorphisms and folds

The *composFold* function may appear to be similar to a *catamorphism* or *fold* (Meijer et al. 1991). However, none of the compositional operations are recursive, as they just apply a given function to the immediate children of the current term. When using a fold, the behavior for each constructor is specified, and the recursion is done by the fold operator. With *composFold*, there is a default behavior for each constructor, and any recursion must be done explicitly.

7.3 Two-level types

Two-level types, as described by Sheard and Pasalic (2004), also address a problem that can lead to repetitive code. Their solution is to break the data type up into two levels, one for the structures that the algorithm manipulates and "a recursive knot-tying level". The problem which the two-level types approach solves is dual to the problem described in this paper: we want to reduce the amount of repeated code when writing many similar functions over the same data type, and they want to reduce the amount of repeated code when writing the same function for many similar data types.

Using the idea of splitting a type into two levels can give us some insight into the relationship between compositional operations and *idiomatic traversals* (the term used by Gibbons (2007) to describe McBride and Paterson's (2008) *traverse* function). We split the Exp type into two levels, making Exp a fixed point of the *structure operator* E. Now *compos* becomes an idiomatic traversal, without changing anything but the type signature (and expanding the catch-all case). The intuition is that E is a container of expressions, and *compos* maps a function over the expressions that it contains. This is only done at the top level, just as a regular map on lists does not descend into any nested lists.

$$\textbf{data}\ \mathsf{E}\ e = \mathsf{EAbs}\ \mathsf{String}\ e \mid \mathsf{EApp}\ e\ e \mid \mathsf{EVar}\ \mathsf{String}$$
$$\textbf{newtype}\ \mathsf{Exp} = \mathsf{Wrap}\ (\mathsf{E}\ \mathsf{Exp})$$
$$\mathit{compos} :: \mathsf{Applicative}\ f \Rightarrow (a \to f\ b) \to \mathsf{E}\ a \to f\ (\mathsf{E}\ b)$$

```
compos f e = case e of
               EAbs x b → pure EAbs ⊛ pure x ⊛ f b
               EApp g h → pure EApp ⊛ f g ⊛ f h
               EVar v   → pure EVar ⊛ pure v
```

We define *composOp*, *composM* and *composFold* as before, but with different types:

```
composOp :: (a → b) → E a → E b
composM :: Monad m ⇒ (a → m b) → E a → m (E b)
composFold :: Monoid o ⇒ (a → o) → E a → o
```

Functions such as *rename* which work on the Exp type now need to use the Wrap constructor, but apart from that, the code is unchanged.

```
rename :: Exp → Exp
rename (Wrap e) = Wrap $ case e of
                           EAbs x b → EAbs ("_" ++ x) (rename b)
                           EVar x   → EVar ("_" ++ x)
                           _        → composOp rename e
```

7.4 The Tree set constructor

Introduction

Petersson and Synek (1989) introduce a set constructor for tree types into Martin-Löf's (1984) intuitionistic type theory. Their tree types are similar to the inductive families in for example Agda (Norell 2007), and, for our purposes, to Haskell's GADTs. The value representation, however, is quite different. There is only one constructor for trees, and it takes as arguments the type index, the data constructor and the data constructor arguments.

Tree types are constructed by the following rule:

TREE SET FORMATION

$$\frac{A : set \qquad B(x) : set[x : A] \qquad C(x,y) : set[x : A, y : B(x)] \qquad d(x,y,z) : A[x : A, y : B(x), z : C(x,y)] \qquad a : A}{Tree(A,B,C,d,a) : set}$$

Here A is the set of names (type indices) of the mutually dependent sets. $B(x)$ is the set of constructors in the set with name x. $C(x,y)$ is the set of argument labels (or selector names) for the arguments of the constructor y in the set with name x. d is a function which assigns types to constructor arguments: for constructor y in the set with name x, $d(x,y,z)$ is the name of the set to which the argument with label z belongs. For simplicity, $\mathcal{T}(a)$ is used below, instead of $Tree(A,B,C,d,a)$.

Tree values are constructed using this rule:

$$\frac{a : A \qquad b : B(a) \qquad c(z) : \mathcal{T}(d(a,b,z))[z : C(a,b)]}{tree(a,b,c) : \mathcal{T}(a)}$$ Tree value introduction

Here a is the name of the set to which the tree belongs, b is the constructor, and c is a function which assigns values to the arguments of the constructor (children of the node), where $c(z)$ is the value of the argument with label z.

Trees are eliminated using the *treerec* constant, with the computation rule:

$$treerec(tree(a,b,c),f) \rightarrow f(a,b,c,\lambda z.\ treerec(c(z),f))$$

Here, f is applied to the tree set name a, the constructor b, the children c, and the results of recursive calls on each of the children. The type of *treerec* is given by:

$$\frac{\begin{array}{l} D(x,t) : set[x : A, t : \mathcal{T}(x)] \qquad a : A \qquad t : \mathcal{T}(a) \\ f(x,y,z,u) : D(x, tree(x,y,z)) \\ \qquad [x : A, y : B(x), z(v) : \mathcal{T}(d(x,y,v))[v : C(x,y)], \\ \qquad\ u(v) : D(d(x,y,v), z(v))[v : C(x,y)]] \end{array}}{treerec(t,f) : D(a,t)}$$ Tree value elimination

Relationship to GADTs

As we have seen above, trees are built using the single constructor *tree*, with the type, constructor, and constructor arguments as arguments to *tree*. We can use this structure to represent GADT values, as long as all children are also trees. Using the constants $l_1 \ldots$ as argument labels for all constructors, we can represent GADT values in the following way:

$$\texttt{b t}_1 \ldots \texttt{t}_n :: \texttt{Tree a} \equiv tree(a,b,\lambda z.\ \text{case } z \text{ of } \{l_1 : t_1; \ldots; l_n : t_n\})$$

For example, the value SDecl TInt (V "foo") :: Tree Stm in our Haskell representation would be represented as the term shown below. We use *"string"* to stand for some appropriate tree representation of a string.

$$\begin{array}{l} tree(Stm, SDecl, \lambda x.\ \text{case } x \text{ of } \{ \\ \qquad\qquad\qquad\qquad\qquad\qquad l_1 : tree(Typ, TInt, \lambda y.\ \text{case } y \text{ of } \{\}); \\ \qquad\qquad\qquad\qquad\qquad\qquad l_2 : tree(Var, V, \lambda y.\ \text{case } y \text{ of } \{l_1 : \textit{"foo"}\}) \\ \}) \end{array}$$

Tree types and compositional operations

We can implement a *composOp*-equivalent in type theory by using *treerec*:

$$composOp(f,t) = treerec(t, \lambda a.\ \lambda b.\ \lambda c.\ \lambda c'.\ tree(a,b,\lambda z.\ f(c(z))))$$

What makes this so easy is that all values have the same representation, and c which contains the child trees is just a function that we can compose with our function f. With this definition, we can use *composOp* like in Haskell. The code below assumes that we have wild card patterns in case expressions, and that ++ is a concatenation operation for whatever string representation we have.

$$\begin{aligned} rename(t) = treerec(t, \lambda a.\ \lambda b.\ \lambda c.\ \lambda c'.\ \text{case } b \text{ of } \{ \\ V : tree(Var, V, \lambda l.\ \text{“\_”} ++ c(l)); \\ \_ : composOp(rename, t) \\ \}) \end{aligned}$$

One advantage over the Haskell solution is that *treerec* is a catamorphism for arbitrary tree types, as it gives us access not only to the original child values (c in the example above), but also to the results of the recursive calls (c' in the example above). This would simplify functions which need to use the results of recursive calls, for example the constant folding example in Section 4.6. As compositional operations are not catamorphisms (see Section 7.2), *composOp* itself does not make use of the c' argument.

7.5 Related work in object-oriented programming

The ComposVisitor class looks deceptively simple, but it has a number of features in what appears to be a novel combination:

- It uses type-parameterized visitor interfaces, which require powerful features such as C++ templates or Java generics. Similar parameterized visitor interfaces can be found in the Loki C++ library (Alexandrescu 2001).
- It is a depth-first traversal combinator whose behavior can be overridden for each concrete class. A similar traversal can be achieved by using the BottomUp and Identity combinators from Visser's (2001) work on visitor combinators, and with the depth-first traversal function in the Boost Graph Library (Lee et al. 2002).
- It allows modification of the data structure in a functional and compositional way. The fact that functional modification is not widely used in imperative object-oriented programming is probably the main reason why this area has not been explored further.

7.6 Nanopass framework for compiler education

The idea of structuring compilers as a large number of simple passes is central to the work on the Nanopass framework for compiler education (Sarkar et al. 2005), a domain-specific language embedded in Scheme. Using the Nanopass framework, a compiler is implemented as a sequence of transformations between a number of intermediate languages, each of which is defined using a set of mutually recursive data types. Transformations are implemented by pattern

matching, and a *pass expander* adds any missing cases, a role similar to that of our *composOp*.

One notable feature of the Nanopass framework is that a language can be declared to inherit from an existing language, with new constructors added or existing ones removed. This makes it possible to give more accurate types to functions which add or remove constructions, without having to define completely separate languages which differ only in the presence or absence of a few constructors. While this is a very useful feature, it is difficult to implement in languages such as Haskell or Java whose notions of data types are more rigid than Scheme's. In Haskell, we model abstract syntax with algebraic datatypes, but Haskell does not allow the extension or restriction of datatypes. In Java, we could add subclasses to encode new constructors, and create new Visitor interfaces for each set of constructors we want to handle, but this would require writing a new ComposVisitor class for each new Visitor interface.

8 Conclusions

We have presented a pattern for easily implementing almost compositional operations over rich data structures such as abstract syntax trees.

We have ourselves started to use this pattern for real implementation tasks, and we feel that it has been very successful. In the compiler for the Transfer language (Bringert 2006) we use a front-end generated by BNFC (Forsberg 2007; Forsberg and Ranta 2006), including a Compos instance for the abstract syntax. The abstract syntax has 70 constructors, and in the (still very small) compiler compositional operations are currently used in 12 places. The typical function that uses compositional operations pattern matches on between 1 and 5 of the constructors, saving hundreds of lines of code. Some of the functions include: replacing infix operator use with function calls, beta reduction, simultaneous substitution, getting the set of variables bound by a pattern, getting the free variables in an expression, assigning fresh names to all bound variables, numbering meta-variables, changing pattern equations to simple declarations using case expressions, and replacing unused variable bindings in patterns with wild cards. Furthermore, we have noticed that using compositional operations to implement a compiler makes it easy to structure it as a sequence of simple steps, without having to repeat large amounts of traversal code for each step. Modifying the abstract syntax, for example by adding new constructs to the front-end language, is also made easier since only the functions which care about this new construct need to be changed. However, using many simple steps is likely to have a negative impact on performance, as a complete traversal is potentially done in every step. This problem could perhaps be ameliorated by developing deforestation techniques (Wadler 1990) for compositional operations.

Acknowledgments

We would like to thank the following people for their comments on earlier versions of this work: Thierry Coquand, Bengt Nordström, Patrik Jansson, Josef Svenningsson, Sibylle Schupp, Marcin Zalewski, Andreas Priesnitz, Markus Forsberg, Alejandro Russo, Thomas Schilling, Andres Löh, the anonymous ICFP and JFP referees, and everyone who offered comments during the talks at the Chalmers CS Winter Meeting, at Galois Connections, and at ICFP 2006. The code in this paper has been typeset using `lhs2TeX`, with help from Andres Löh and Jeremy Gibbons. This work has been partly funded by the EU TALK project, IST-507802.

References

Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, Indianapolis, February 2001. ISBN 0201704315.

Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, second edition, October 2002. ISBN 052182060X.

Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, December 1997. ISBN 0521582741.

Lennart Augustsson and Kent Petersson. Silly type families. `http://www.cs.pdx.edu/~sheard/papers/silly.pdf`, 1994. URL `http://www.cs.pdx.edu/~sheard/papers/silly.pdf`.

Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In Craig Chambers, editor, *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Vancouver, BC*, pages 183–200, New York, NY, USA, 1998. ACM. doi: 10.1145/286936.286957.

Björn Bringert. The Transfer programming language, 2006. `http://www.cs.chalmers.se/Cs/Research/Language-technology/GF/doc/transfer.html`.

Catarina Coquand and Thierry Coquand. Structured type theory. In *Workshop on Logical Frameworks and Meta-languages (LFM'99), Paris, France*, September 1999. URL `http://citeseer.ist.psu.edu/coquand99structured.html`.

Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, July 1994. doi: 10.1007/BF01211308.

Markus Forsberg. *Three Tools for Language Processing: BNF Converter, Functional Morphology, and Extract*. PhD thesis, Göteborg University and

Chalmers University of Technology, September 2007. URL http://www.cs.chalmers.se/~markus/phd2007_print_version.pdf.

Markus Forsberg and Aarne Ranta. BNF Converter homepage, 2006. http://www.cs.chalmers.se/~markus/BNFC/.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0201633612. URL http://portal.acm.org/citation.cfm?id=186897.

Jeremy Gibbons. Datatype-Generic Programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 1–71, Heidelberg, November 2007. Springer. doi: 10.1007/978-3-540-76786-2_1.

Jeremy Gibbons and Bruno C. Oliveira. The Essence of the Iterator Pattern. In Conor McBride and Tarmo Uustalu, editors, *Workshop on Mathematically Structured Functional Programming (MSFP 2006), Kuressaare, Estonia*, Electronic Workshops in Computing (eWiC), Swindon, UK, July 2006. British Computer Society. URL http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/iterator-msfp.pdf.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley Professional, Indianapolis, third edition, July 2005. ISBN 0321246780.

Ralf Hinze. Generics for the masses. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, volume 39, pages 236–243, New York, September 2004. ACM Press. doi: 10.1145/1016850.1016882.

Ralf Hinze, Andres Löh, and Bruno C. D. S. Oliveira. "Scrap Your Boilerplate" Reloaded. In Masami Hagiya and Philip Wadler, editors, *8th International Symposium on Functional and Logic Programming (FLOPS 2006), Fuji-Susono, Japan*, volume 3945 of *Lecture Notes in Computer Science*, pages 13–29, Heidelberg, 2006. Springer. doi: 10.1007/11737414_3.

Patrik Jansson and Johan Jeuring. PolyP - a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 470–482, New York, NY, USA, 1997. ACM Press. ISBN 0897918533. doi: 10.1145/263699.263763.

Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136, Heidelberg, 1995. Springer. ISBN 3540594515. doi: 10.1007/3-540-59451-5_4.

Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003), New Orleans, LA, USA*, pages 26–37, New York, NY, USA, January 2003. ACM. doi: 10.1145/604174.604179.

Lie-Quan Lee, Andrew Lumsdaine, and Jeremy G. Siek. *The Boost graph library: user guide and reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0-201-72914-8.

Daniel Leivant. Polymorphic type inference. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 88–98, New York, NY, USA, 1983. ACM. ISBN 0897910907. doi: 10.1145/567067.567077.

Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984.

Conor McBride and Ross Paterson. Applicative Programming with Effects. *Journal of Functional Programming*, 18(1):1–13, January 2008. doi: 10.1017/S0956796807006326.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144, Heidelberg, 1991. Springer. ISBN 0387543961. doi: 10.1007/3540543961_7.

Neil Mitchell and Stefan O'Rear. Data.Derive: A User Manual, August 2007. `http://www.cs.york.ac.uk/fp/darcs/derive/derive.htm`.

Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, USA, July 1990. ISBN 0198538146.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, 2007. URL `http://www.cs.chalmers.se/~ulfn/papers/thesis.pdf`.

Kent Petersson and Dan Synek. A set constructor for inductive sets in Martin-Löf's type theory. In *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 128–140, Heidelberg, 1989. Springer. doi: 10.1007/BFb0018349.

Simon Peyton Jones. The GHC Commentary, August 2007. `http://hackage.haskell.org/trac/ghc/wiki/Commentary`.

Simon Peyton Jones. The Haskell 98 Language. *Journal of Functional Programming*, 13(1):1–146, 2003a.

Simon Peyton Jones. The Haskell 98 Libraries. *Journal of Functional Programming*, 13(1):149–240, 2003b.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 50–61, New York, NY, USA, 2006. ACM Press. ISBN 1595933093. doi: 10.1145/1159803.1159811.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, 2007. doi: 10.1017/S0956796806006034.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004. ISSN 0956-7968. doi: 10.1017/S0956796803004738.

Dipanwita Sarkar, Oscar Waddell, and Kent R. Dybvig. EDUCATIONAL PEARL: A Nanopass framework for compiler education. *Journal of Functional Programming*, 15(5):653–667, 2005. doi: 10.1017/S0956796805005605.

Tim Sheard and Emir Pasalic. Two-level types and parameterized modules. *Journal of Functional Programmming*, 14(5):547–587, September 2004. ISSN 0956-7968. doi: 10.1017/S095679680300488X.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell '02: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM. ISBN 1581136056. doi: 10.1145/581690.581691.

Joost Visser. Visitor combination and traversal control. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, volume 36, pages 270–282, New York, NY, USA, November 2001. ACM Press. ISBN 1581133359. doi: 10.1145/504282.504302.

Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990. doi: 10.1016/0304-3975(90)90147-A.

Noel Winstanley, Malcom Wallace, and John Meacham. The DrIFT homepage, 2007. `http://repetae.net/~john/computer/haskell/DrIFT/`.