A Functional Semantics of Attribute Grammars

Kevin Backhouse

Oxford University Computing Laboratory

Abstract. A definition of the semantics of attribute grammars is given, using the lambda calculus. We show how this semantics allows us to prove results about attribute grammars in a calculational style. In particular, we give a new proof of Chirica and Martin's result [6], that the attribute values can be computed by a structural recursion over the tree. We also derive a new *definedness test*, which encompasses the traditional closure and circularity tests. The test is derived by abstract interpretation.

1 Introduction

This paper introduces a new definition of the semantics of attribute grammars (AGs), which we believe is easier to manipulate mathematically. The semantics is based on the lambda calculus, so a calculational proof style is possible. We illustrate the power of the semantics by proving Chirica and Martin's result [6]: that AGs can be evaluated by structural recursion. More importantly though, we show that the semantics is an excellent basis from which to derive tools and algorithms. We illustrate this by deriving a new *definedness test* for AGs. Our original goal was to derive Knuth's circularity test [12,13], but we were surprised to discover a new test that is actually slightly more powerful than Knuth's.

Notation and Semantic Framework. Our notation is based on the notation of the functional language Haskell [4]. Although the definitions given in Section 2 are valid Haskell, we do not restrict ourselves to Haskell's lazy semantics. Instead we merely assume that the computations are over a semantic domain in which the least fixed points exist. This is so as not to rule out work such as Farrow's [9], in which non-flat domains are used to great effect. For example, Farrow uses the set domain to do data flow analysis. Throughout the paper, we use a number of standard functions such as *fst* and *map*. These are defined in the appendix. Our definition of sequences, which is a little unusual, is also given in the appendix.

2 The Semantics of Attribute Grammars

In this section, we define the semantics of AGs as a least fixed point computation. Our approach is to represent abstract syntax trees as *rose trees* and then define the semantics using functions over rose trees. Throughout, we illustrate our definitions with Bird's *repmin* problem [3]. As noted by Kuiper and Swierstra [14], repmin can be easily written as an attribute grammar. The input to repmin is a binary tree of integers. Its output is a tree with identical structure, but with every leaf value replaced by the minimum leaf value of the input tree.

J.-P. Katoen and P. Stevens (Eds.): TACAS 2002, LNCS 2280, pp. 142–157, 2002.

2.1 Abstract Syntax Trees as Rose Trees

We use rose trees (see appendix) to represent abstract syntax trees.

$$\textbf{type}\ \mathit{AST}\ =\ \mathit{Rose}\ \mathit{ProdLabel}$$

The type *ProdLabel* is a sum type representing the different productions of the grammar. In repmin, The abstract syntax tree is a binary tree, so *ProdLabel* is:

$$\textbf{data}\ \mathit{ProdLabel}\ =\ \mathit{Root} \mid \mathit{Fork} \mid \mathit{Leaf}\ \mathit{Int}$$

A single *Root* production appears at the top of the tree. It has one child, which is the binary tree. *Fork* nodes have two children and *Leaf* nodes have zero. Note that the type *ProdLabel* can contain terminal information such as integer constants and strings, but it does not contain non-terminal information. The non-terminals of a production are represented as children in the rose tree.

2.2 Attributes

A drawback of our use of rose trees is that every node in the tree must have the same set of attributes. That is, if leaf nodes have a *code* attribute then root nodes have a *code* attribute too. However, attributes can be left undefined and our definedness test ensures that undefined attributes are not used. Repmin has one inherited attribute for the global minimum leaf value and two synthesised attributes: the minimum leaf value for the current subtree and the output tree for the current subtree. So we define the following datatypes:[1]

$$\begin{aligned}&\textbf{type}\ \mathit{Inh}\ &&=\mathit{Int}_\perp\\&\textbf{type}\ \mathit{Syn}\ &&=(\mathit{Int}_\perp,\ \mathit{AST}_\perp)\end{aligned}$$

In an AG with more attributes, we might use records or lookup tables, rather than tuples to store the attributes. Our definition of AGs below uses *polymorphism* to leave this choice open.

2.3 Attribute Grammars as Functions

In the following definition of an attribute grammar, the inherited and synthesised attributes are represented by the type parameters α and β, respectively. As explained above, this means that we are not bound to a particular implementation of attributes. (*Seq* is defined in the appendix.)

$$\begin{aligned}&\textbf{type}\ \mathit{AG}\ \alpha\ \beta\ &&=\mathit{ProdLabel}\ \rightarrow\ \mathit{SemRule}\ \alpha\ \beta\\&\textbf{type}\ \mathit{SemRule}\ \alpha\ \beta\ &&=(\alpha,\ \mathit{Seq}\ \beta)\ \rightarrow\ (\beta,\ \mathit{Seq}\ \alpha)\end{aligned}$$

[1] The $\perp$ subscript on the types indicates that we have lifted them to a flat domain by adding $\perp$ and $\top$ elements. We use a similar informal notation for lifting functions and values: if x has type A, then $x_\perp$ has type $A_\perp$ and if f has type $A \rightarrow B$, then $f_\perp$ has type $A_\perp \rightarrow B_\perp$.

This definition states that an AG is a set of semantic rules, indexed by the productions of the grammar. Our concept of a semantic rule deviates slightly from tradition. Traditionally, there is one semantic rule per attribute. In our model, a single semantic rule defines all the attribute values. It is a function from the "input" attributes to the "output" attributes. The input attributes are the inherited attributes of the parent node and the synthesised attributes of the children. The output attributes are the synthesised attributes of the parent and the inherited attributes of the children. The fact that we are using a single function does not mean that the attribute values have to be computed simultaneously. For example, in Haskell [4] attribute values would be evaluated on demand, due to the lazy semantics. In Farrow's model [9], attribute values are computed iteratively, so the function would be called several times.

Repmin. Repmin is encoded as a value of type AG as follows:

$$
\begin{array}{ll}
\mathit{repAG} & :: \mathit{AG\ Inh\ Syn} \\
\mathit{repAG\ Root}\ (\_,\ \mathit{syns}) & = (\ (\bot,\ \mathit{Node}_\bot\ \mathit{Root}\ [\mathit{snd\ syns}_1]),\ [\mathit{fst\ syns}_1]\ ) \\
\mathit{repAG\ Fork}\ (\mathit{gmin},\ \mathit{syns}) & = (\ (\ \mathit{min}_\bot\ (\mathit{fst\ syns}_1)\ (\mathit{fst\ syns}_2), \\
 & \qquad \mathit{Node}_\bot\ \mathit{Fork}\ [\mathit{snd\ syns}_1,\ \mathit{snd\ syns}_2]\ ), \\
 & \quad [\mathit{gmin},\ \mathit{gmin}]\ ) \\
\mathit{repAG}\ (\mathit{Leaf}\ k)\ (\mathit{gmin},\ \mathit{syns}) & = (\ (k_\bot,\ \mathit{Node}_\bot\ (\mathit{Leaf}_\bot\ \mathit{gmin})\ [\,]),\ [\,]\ )
\end{array}
$$

This should be read as follows. The local minimum of the root production is undefined, because it will remain unused. The output tree of the root production is equal to the output tree of its subtree. In the root production, the local minimum of the subtree becomes the inherited global minimum attribute. (The *gmin* attribute of the root node is ignored.) In the fork production, the local minimum is the minimum of the two local minima of the subtrees. The output tree is built from the output trees of the two subtrees and the global minimum is passed unchanged to the subtrees. The local minimum in the leaf production equals the value at the leaf and the new tree is a leaf node with the global minimum as its value.

Semantic Trees. Given an AST and an AG, we can compute a *semantic tree*, which is a tree containing semantic functions. This is done by simply mapping the AG over the tree:

$$
\begin{array}{l}
\textbf{type}\ \mathit{SemTree}\ \alpha\ \beta\ =\ \mathit{Rose}\ (\mathit{SemRule}\ \alpha\ \beta) \\
\mathit{mkSemTree}\ ::\ \mathit{AG}\ \alpha\ \beta\ \to\ \mathit{AST}\ \to\ \mathit{SemTree}\ \alpha\ \beta \\
\mathit{mkSemTree}\ =\ \mathit{mapRose}
\end{array}
$$

We will often work with semantic trees rather than abstract syntax trees, because it makes some of our definitions and proofs more concise.

2.4 Shifting Attributes Up and Down the Tree

The goal of an attribute grammar evaluator is to produce a tree containing the attribute values. In other words we wish to produce a value of type $Rose\ (\alpha, \beta)$, where α and β are the types of the inherited and synthesised attributes. However, this format is not easily compatible with the semantic rules defined earlier. Instead we use the following two formats to represent annotated trees:

$$\begin{aligned}&\textbf{type}\ InputTree\ \alpha\ \beta &&= Rose\ (\alpha,\ Seq\ \beta)\\ &\textbf{type}\ OutputTree\ \alpha\ \beta &&= Rose\ (\beta,\ Seq\ \alpha)\end{aligned}$$

In the *input* format, the synthesised attributes of a node are stored by the node's parent. The parent uses a sequence to store the synthesised attributes of all its children. Similarly, in the *output* format, the inherited attributes of a node are stored by the node's parent. The output tree is the result of applying the semantic tree to the input tree. Note that in the input format, the synthesised attributes of the root node are not stored. Similarly, in the output format the inherited attributes are not stored. Therefore, this data needs to be stored separately.

The *shift* function is used to convert from the output to the input format. It encapsulates the idea that inherited attributes are passed down the tree and synthesised attributes are passed up. Note that *shift* merely moves attributes around; the calculation of the attribute values is done by the semantic tree.

$$\begin{aligned}&shift &&:: \alpha \rightarrow OutputTree\ \alpha\ \beta \rightarrow InputTree\ \alpha\ \beta\\ &shift\ a\ (Node\ (b, as)\ ts) &&= Node\ (a, bs)\ (zipWith\ shift\ as\ ts)\\ &\quad \textbf{where}\ bs &&= map\ (fst \circ root)\ ts\end{aligned}$$

Note that the inherited attributes of the root node are the first parameter of *shift*. The inherited attributes of the root node are always an external parameter to an attribute grammar.

2.5 The Semantics as a Least Fixed Point Computation

We explained above how we can produce a *semantic tree* by mapping an AG over an AST. Given a semantic tree *semtree* and the inherited attributes a of the root node, we expect the following equations to hold:

$$inputTree = shift\ a\ outputTree \tag{1}$$

$$outputTree = appRose\ semtree\ inputTree \tag{2}$$

As we discussed above, the attributes of the tree can be stored either in the input or the output format. Equation (1) states that the input tree can be computed from the output tree with the *shift* function. Equation (2) says that the output tree can be computed by applying the semantic tree to the input tree. Equations (1) and (2) form a pair of simultaneous equations that can be used to compute *inputTree* and *outputTree*. Following Chirica and Martin [6], we define the semantics of attribute grammars to be the *least* solution of the equations. This choice has no effect if the attributes are not circularly defined, because the

solution to the equations is unique. If our domains are flat, then circularly defined attributes will evaluate to $\bot$. Farrow [9] discusses some interesting applications of working with non-flat domains. The least solution can be computed with the following function:

$$\begin{array}{ll} \mathit{eval} & :: \mathit{SemTree}\ \alpha\ \beta\ \to\ \alpha\ \to\ \mathit{OutputTree}\ \alpha\ \beta \\ \mathit{eval}\ \mathit{semtree}\ a & = \mu\ (\mathit{appRose}\ \mathit{semtree}\ \circ\ \mathit{shift}\ a) \end{array}$$

This function is our definition of the semantics of attribute grammars. Please note that we do not propose it as an efficient implementation though! In the following section, we define a more convenient version called *io* and demonstrate it on the repmin problem (Section 3.4).

3 AG Semantics as a Structural Recursion

The *eval* function given above computes attribute values for the entire tree. In practice, we often only want to know the values of the synthesised attributes of the root node of the tree. Therefore, it is convenient to define the function *io*:

$$\begin{array}{ll} \mathit{io} & :: \mathit{SemTree}\ \alpha\ \beta\ \to\ \alpha\ \to\ \beta \\ \mathit{io}\ \mathit{semtree}\ a & = \mathit{fst}\ (\mathit{root}\ (\mathit{eval}\ \mathit{semtree}\ a)) \end{array}$$

The name *io* stands for input-output. It converts the semantic tree into a function from inherited attributes to synthesised attributes.

Johnsson [11] showed that AGs can be viewed as an idiom of lazy functional programming. Using the idea that a semantic tree is simply a function from inherited to synthesised attributes, he recursively builds the input-output function for the entire AST. The idea that input-output functions can be computed by structural recursion was also stated much earlier by Chirica and Martin [6]. We state the idea formally with the following theorem:

Theorem 1. *The following definition of io is equivalent to the one given above:*

$$\mathit{io} = \mathit{foldRose}\ \mathit{knit}$$

where:

$$\begin{array}{ll} \mathit{knit} & :: (\mathit{SemRule}\ \alpha\ \beta,\ \mathit{Seq}\ (\alpha\ \to\ \beta))\ \to\ (\alpha\ \to\ \beta) \\ \mathit{knit}\ (\mathit{semrule}, \mathit{fs})\ a & = \mathit{fst}\ (\mu\ \langle\ b, \mathit{as}\ \bullet\ \mathit{semrule}\ (a,\ \mathit{appSeq}\ \mathit{fs}\ \mathit{as})\rangle) \end{array}$$

The definition of *knit* is rather unintuitive, but it has appeared in different forms in many papers on AGs, including one of our own [8]. Other examples are Gondow & Katayama [10] and Rosendahl [18]. Very briefly, *as* and *appSeq fs as* are respectively the inherited and synthesised attributes of the subtrees. Their values are mutually dependent (using *semrule*), so they are computed by a least fixed point computation. For a better understanding of *knit*, see Johnsson [11].

Our proof is essentially the same as the proof given by Chirica and Martin [6], although our notation is rather different. The most important step is to use the mutual recursion rule and we will start by proving a lemma, that adapts the mutual recursion rule to rose trees. Another important step is to use the abstraction rule though. The use of the abstraction rule is the motivation for our use of sequences rather than lists throughout this document.

3.1 The Mutual Recursion Rule on Rose Trees

The most crucial step in our proof of Theorem 1 is to use the *mutual recursion rule* (see the appendix). This theorem allows us to factorise least fixed point computations. In this section, we shall show how it can be applied to rose trees. We do this by proving the following lemma:

Lemma 1. *If f is a monotonic function of type Rose $A \rightarrow$ Rose A, then:*

$$root\ (\mu\, f)\ =\ \mu\, \langle\, x\ \bullet\ g_1\ (x,\ \mu\, \langle\, xs\ \bullet\ g_2\ (x, xs)\rangle)\rangle$$

where:

$$\begin{aligned} g_1\ &= fst\ \circ\ g \\ g_2\ &= snd\ \circ\ g \\ g\ &= split\ \circ\ f\ \circ\ merge \end{aligned}$$

Proof:

$$\begin{aligned} & root\ (\mu\ f) \\ =\ & \quad \{\ merge \circ split = id\ \} \\ & root\ (\mu\ (merge \circ split \circ f)) \\ =\ & \quad \{\ \text{rolling rule (see the appendix)}\ \} \\ & root\ (merge\ (\mu\ (split \circ f \circ merge))) \\ =\ & \quad \{\ root \circ merge = fst, \text{Definition } g\ \} \\ & fst\ (\mu\ g) \\ =\ & \quad \{\ \text{mutual recursion rule (see the appendix)}\ \} \\ & \mu\ \langle x \bullet g_1\ (x,\ \mu\ \langle xs \bullet g_2\ (x, xs)\rangle)\rangle \end{aligned}$$

3.2 Proving Theorem 1

We now prove Theorem 1, deriving the definition of *knit* in the process. First we apply a general result about folds:

$$\begin{aligned} & io = foldRose\ knit \\ \equiv\ & \quad \{\ \text{Universal property of fold, Bird \& de Moor [5, page 46]}\ \} \\ & \forall \langle s, ss, a \bullet\ io\ (Node\ s\ ss)\ a = knit\ (s, map\ io\ ss)\ a\ \rangle \end{aligned}$$

We continue by manipulating the expression $io\ (Node\ s\ ss)\ a$ until we obtain an expression from which it is easy to derive the definition of *knit*:

$$\begin{aligned} & io\ (Node\ s\ ss)\ a \\ =\ & \quad \{\ \text{Definition } io\ \} \\ & fst\ (root\ (\mu\ (appRose\ (Node\ s\ ss) \circ shift\ a))) \end{aligned}$$

We can apply the mutual recursion rule on rose trees (Lemma 1) to this expression. By simply expanding the definitions of g_1 and g_2, as specified in Lemma 1, we find that:

$$g_1\ ((b, as),\ ts)\ = s\ (a,\ map\ (fst \circ root)\ ts)$$

$$g_2\ ((b, as),\ ts)\ = zipWith\ appRose\ ss\ (zipWith\ shift\ as\ ts)$$

Therefore, Lemma 1 implies that $io\ (Node\ s\ ss)\ a$ is equal to:

$$fst\ (\mu\ \langle b, as \bullet\ g_1\ ((b, as),\ \mu\ \langle ts \bullet\ g_2\ ((b, as), ts)\rangle)\rangle)$$

(With g_1 and g_2 as defined above.) Consider the sub-expression containing g_2:

$$\begin{array}{ll}
 & \mu\ \langle ts \bullet g_2\ ((b, as), ts)\rangle \\
= & \quad \{\ \text{Definition } g_2\ \} \\
 & \mu\ \langle ts \bullet zipWith\ appRose\ ss\ (zipWith\ shift\ as\ ts)\rangle \\
= & \quad \{\ \text{Claim (see the appendix)}\ \} \\
 & \mu\ \langle ts \bullet appSeq\ (zipWith\ (\circ)\ (map\ appRose\ ss)\ (map\ shift\ as))\ ts\rangle \\
= & \quad \{\ \text{abstraction rule (see the appendix)}\ \} \\
 & \langle k \bullet\ \mu\ (zipWith\ (\circ)\ (map\ appRose\ ss)\ (map\ shift\ as)\ k)\rangle \\
= & \quad \{\ \text{Definitions } zipWith \text{ and } map\ \} \\
 & \langle k \bullet\ \mu\ (appRose\ (ss\ k)\ \circ\ shift\ (as\ k))\rangle \\
= & \quad \{\ \text{Definition } eval\ \} \\
 & \langle k \bullet\ eval\ (ss\ k)\ (as\ k)\rangle
\end{array}$$

We continue with the main expression:

$$\begin{array}{ll}
 & fst\ (\mu\ \langle b, as \bullet g_1\ ((b, as),\ \mu\ \langle ts \bullet g_2\ ((b, as), ts)\rangle)\rangle) \\
= & \quad \{\ \text{Substitute the new sub-expression}\ \} \\
 & fst\ (\mu\ \langle b, as \bullet g_1\ ((b, as),\ \langle k \bullet eval\ (ss\ k)\ (as\ k)\rangle)\rangle) \\
= & \quad \{\ \text{Definition } g_1\ \} \\
 & fst\ (\mu\ \langle b, as \bullet s\ (a, map\ (fst \circ root)\ \langle k \bullet eval\ (ss\ k)\ (as\ k)\rangle)\rangle) \\
= & \quad \{\ \text{Definitions } map \text{ and } io\ \} \\
 & fst\ (\mu\ \langle b, as \bullet s\ (a, \langle k \bullet io\ (ss\ k)\ (as\ k)\rangle)\rangle) \\
= & \quad \{\ \text{Definitions } appSeq \text{ and } map\ \} \\
 & fst\ (\mu\ \langle b, as \bullet s\ (a, appSeq\ (map\ io\ ss)\ as)\rangle) \\
= & \quad \{\ \text{Definition } knit\ \} \\
 & knit\ (s, map\ io\ ss)\ a
\end{array}$$

In the final step, we have derived the definition of *knit*.

3.3 A More Convenient Version of the *io* Function: *trans*

The *io* function operates on semantic trees. It is usually more convenient to operate on abstract syntax trees, so we define the *trans* function:

$$\begin{aligned} &\mathit{trans} &&:: \mathit{AG}\ \alpha\ \beta \rightarrow \mathit{AST} \rightarrow \alpha \rightarrow \beta \\ &\mathit{trans}\ \mathit{ag} &&= \mathit{io} \circ \mathit{mkSemTree}\ \mathit{ag} \end{aligned}$$

In the next section, it is important that *trans* can be written as a fold:

$$\mathit{trans}\ \mathit{ag} = \mathit{foldRose}\ (\mathit{knit} \circ (\mathit{ag} \times \mathit{id}))$$

The proof is a simple application of fold-map fusion [4, page 131].

3.4 Repmin as a Recursive Function

Let us expand *trans repAG* to demonstrate that it is equivalent to the functional program for repmin. Expanding the definitions of *trans* (above), *foldRose* (appendix), *knit* (page 146) and *repAG* (page 144), we obtain:

$$\begin{aligned} &\mathit{repmin} &&:: \mathit{AST} \rightarrow \mathit{Inh} \rightarrow \mathit{Syn} \\ &\mathit{repmin}\ (\mathit{Node}\ \mathit{Root}\ [t])\ \_ &&= (\bot,\ \mathit{Node}_\bot\ \mathit{Root}\ [\mathit{nt}]) \\ &&&\quad \mathbf{where}\ (\mathit{gm}, \mathit{nt}) = \mathit{repmin}\ t\ \mathit{gm} \\ &\mathit{repmin}\ (\mathit{Node}\ \mathit{Fork}\ [t_1, t_2])\ \mathit{gm} &&= (\mathit{min}_\bot\ \mathit{lm}_1\ \mathit{lm}_2,\ \mathit{Node}_\bot\ \mathit{Fork}\ [\mathit{nt}_1,\ \mathit{nt}_2]) \\ &&&\quad \mathbf{where}\ (\mathit{lm}_1, \mathit{nt}_1) = \mathit{repmin}\ t_1\ \mathit{gm} \\ &&&\qquad\quad (\mathit{lm}_2, \mathit{nt}_2) = \mathit{repmin}\ t_2\ \mathit{gm} \\ &\mathit{repmin}\ (\mathit{Node}\ (\mathit{Leaf}\ k)\ \_)\ \mathit{gm} &&= (k_\bot,\ \mathit{Node}_\bot\ (\mathit{Leaf}_\bot\ \mathit{gm})\ [\,]) \end{aligned}$$

This is indeed very similar to the program given by Bird [3]. Of course Bird's emphasis is on the recursive definition of *gm* at the root of the tree, which is valid in a language with lazy evaluation.

4 The Definedness Test

When Knuth [12] introduced attribute grammars, he also gave algorithms for testing them for closure and circularity. Closure is the property that there are no 'missing' semantic rules for attributes. An AG is circular if there exists an input tree on which the attributes are circularly defined. We present a new algorithm, which is very similar to the circularity test, but encompasses the roles of both the closure and the circularity tests. It computes equivalent results to the circularity test and improves upon the closure test (which is based on a very shallow analysis of the dependency structure) by reducing the number of false negatives. In other work on extending attribute grammars [19] we have found this improvement essential.

In our model, an undefined attribute is an attribute with the value $\bot$. Similarly, circularly defined attributes over a flat domain will evaluate to $\bot$, because

our semantics is a least fixed point semantics.[2] So in our model, the goal of both the closure and the circularity tests is to detect attributes which will evaluate to $\bot$. Our *definedness test* accomplishes both of these goals by analysing *strictness*. This concept is very similar to the concept of "dependence" used in the circularity test. A function f is *strict* if $f\ \bot = \bot$. In our model, an undefined attribute is an attribute with the value $\bot$, so a strict function 'depends' on its parameter. However, the expression $f\ x$ is said to depend on x even if f is not strict, so the two concepts are not identical. Luckily, semantic rules in a traditional AG are always strict in the attributes that they depend on.

Our presentation of the circularity test consists of two parts. In the first part we explain how the strictness analysis is done as an abstract interpretation. In the second part we explain how the AG can be statically tested for circularity on *all* possible input trees.

4.1 Strictness Analysis and Abstract Interpretation

Strictness analysis is very important for the optimisation of lazy functional programs. Peyton Jones [16, pages 380-395] explains this and gives a good introduction to *abstract interpretation*, which is the technique used to perform the analysis. The use of strictness analysis in AGs has been studied before by Rosendahl [18]. However, Rosendahl's emphasis is on the optimisation of AGs, rather than definedness.

The prerequisite for abstract interpretation (first introduced by Cousot & Cousot [7]) is a *Galois Connection* between the "concrete" domain $(A, \leq)$ and the "abstract" domain $(A^{\#}, \preceq)$. For an introduction to Galois Connections, see Aarts [1]. Briefly, there should be an *abstraction* function *abs* of type $A \to A^{\#}$ and a *concretisation* function *con* of type $A^{\#} \to A$. The "connection" is:

$$\forall\langle x, y \bullet \ abs\ x \preceq y \quad \equiv \quad x \leq con\ y\rangle$$

The Galois Connection used in strictness analysis is between any domain $(A, \leq)$ and the boolean domain $(Bool, \Rightarrow)$:

$$\begin{array}{ll} abs_A \quad :: A \to Bool & con_A \quad :: Bool \to A \\ abs_A\ x = (x \neq \bot) & con_A\ b = \textbf{if}\ b\ \textbf{then}\ \top\ \textbf{else}\ \bot \end{array}$$

The idea is that in the abstract domain, $\bot$ is represented by *False* and other values by *True*. Given a function f, we derive an abstract version $f^{\#}$ that models the strictness of f using booleans. Suppose for example that f has type $A \to B$. Then $f^{\#}$ will have type $Bool \to Bool$. The result of $f^{\#}$ is *False* if and only if the result of f is $\bot$. Mathematically, this relationship between f and $f^{\#}$ is:[3]

$$\forall\langle x \bullet \ abs_B\ (f\ x) \quad = \quad f^{\#}\ (abs_A\ x)\ \rangle \tag{3}$$

[2] If the domain is non-flat, then circular attributes may not evaluate to $\bot$, but in this situation circularity is usually intended rather than erroneous. Farrow [9] suggests some useful applications of circular attributes over non-flat domains.

[3] Abstract interpretations are often only able to give *conservative* results, so the equation is an inquality: $abs\ (f\ x) \leq f^{\#}\ (abs\ x)$. However, in most AGs such as repmin, an exact result can be given.

Often, if this equation has a solution, it is the function $f^{\#} = abs_B \circ f \circ con_A$. We will use this technique when we discuss repmin below.

Repmin. Let us apply strictness analysis to repmin. First we must define abstract versions of the types *Inh* and *Syn*, defined on page 143:

$$\begin{aligned}&\textbf{type } Inh^{\#} &&= Bool\\ &\textbf{type } Syn^{\#} &&= (Bool,\ Bool)\end{aligned}$$

Inh and $Inh^{\#}$ are Galois Connected, as are *Syn* and $Syn^{\#}$:

$$\begin{aligned}absInh &= abs_{Int} & absSyn &= abs_{Int} \times abs_{AST}\\ conInh &= con_{Int} & conSyn &= con_{Int} \times con_{AST}\end{aligned}$$

Now we need to derive an abstract version of *repAG*, the attribute grammar for repmin, defined on page 144. The type of *repAG* is *AG Inh Syn*, so $repAG^{\#}$ should have type $AG\ Inh^{\#}\ Syn^{\#}$. Following the suggestion above, we define:

$$repAG^{\#}\ p\ =\ (absSyn \times map\ absInh) \circ repAG\ p \circ (conInh \times map\ conSyn)$$

A simple calculation reveals that *repAG* and $repAG^{\#}$ satisfy an equation similar to (3), which is required by Theorem 2. An equivalent definition of $repAG^{\#}$ is:

$$\begin{aligned}&repAG^{\#} &&:: AG\ Inh^{\#}\ Syn^{\#}\\ &repAG^{\#}\ Root\ (\_,\ syns) &&= ((False,\ snd\ syns_1),\ [fst\ syns_1])\\ &repAG^{\#}\ Fork\ (gmin,\ syns) &&= ((\ fst\ syns_1\ \wedge\ fst\ syns_2,\\ &&&\qquad snd\ syns_1\ \wedge\ snd\ syns_2),\ [gmin,\ gmin])\\ &repAG^{\#}\ (Leaf\ k)\ (gmin,\ \_) &&= ((True,\ gmin),\ [\,])\end{aligned}$$

The reader should compare this definition with the definition of *repAG*, given on page 144. Where *repAG* computes the value $\bot$, this function computes *False*.

Using the Abstract Interpretation. Now that we have $ag^{\#}$, the abstract interpretation of *ag*, what happens when we evaluate an abstract syntax tree using it? Our hope is that the result of evaluating the tree with $ag^{\#}$ can be used to predict something about the result of evaluating the tree with *ag*. The following theorem is what we need:

Theorem 2. *Suppose we are given an attribute grammar ag and its abstract interpretation* $ag^{\#}$*. By definition, they are related as follows, for all p:*

$$(absSyn \times map\ absInh) \circ ag\ p \quad = \quad ag^{\#}\ p \circ (absInh \times map\ absSyn)$$

Then, for all t:

$$absSyn \circ trans\ ag\ t \quad = \quad trans\ ag^{\#}\ t \circ absInh$$

This theorem could be worded as: if $ag^{\#}$ is the abstract interpretation of *ag*, then *trans* $ag^{\#}$ is the abstract interpretation of *trans ag*. The theorem can be proved manually using fixpoint fusion or fixpoint induction, but we have discovered that it follows immediately from the polymorphic type of *trans*. This is the topic of a forthcoming paper [2], which presents a result based on Reynold's parametricity theorem [17], similar to Wadler's "Theorems for free" [20].

4.2 Computing the Strictness for All Possible Trees

In general an attribute grammar will have n inherited attributes and m synthesised attributes. Therefore, the types $Inh^{\#}$ and $Syn^{\#}$ will be:

$$\begin{aligned}&\textbf{type } Inh^{\#} &&= (Bool,\ \ldots,\ Bool)\\ &\textbf{type } Syn^{\#} &&= (Bool,\ \ldots,\ Bool)\end{aligned}$$

This means that there are 2^n different values of type $Inh^{\#}$ and 2^m of type $Syn^{\#}$. Consequently, there are only a finite number of functions of type $Inh^{\#} \to Syn^{\#}$. So even though there are an infinite number of input trees, $trans\ ag^{\#}$ can only have a finite number of values. In this section we derive an algorithm, very similar to the traditional circularity test algorithm, for computing those values in a finite amount of time.

The Grammar. Until this point we have ignored the fact that abstract syntax trees are constrained by a *grammar*. We are using rose trees to represent ASTs, which means that we are not prevented from building grammatically incorrect trees. The definedness test depends crucially on the grammar, so let us define it.

A grammar consists of a set of *symbols* and an associated set of *productions*. Just like productions, we represent the symbols with a simple enumeration datatype. In repmin, there are just two symbols:

$$\textbf{data } Symbol\ =\ Start \mid Tree$$

Every symbol in the grammar is associated with zero or more productions. Every production has a sequence of symbols on its right hand side. We represent this as a *partial function* with the following type:

$$grammar\ ::\ (ProdLabel,\ Seq\ Symbol) \nrightarrow Symbol$$

This function is partial, because it only accepts inputs where the sequence of symbols matches the production. For example, the grammar for repmin is:

$$\begin{aligned}&grammar\ (Root,\ [Tree]) &&= Start\\ &grammar\ (Fork,\ [Tree, Tree]) &&= Tree\\ &grammar\ (Leaf\ \_,\ [\,]) &&= Tree\end{aligned}$$

The following function checks whether a tree is grammatically correct:

$$foldRose\ grammar\ ::\ AST \nrightarrow Symbol$$

This function is also partial. It is only defined on grammatically correct trees.

Generating Grammatically Correct Trees. We can generate the set of all grammatically correct trees by *inverting* the grammar:

$$\begin{aligned}&trees\ ::\ Symbol \leftrightarrow AST\\ &trees\ =(foldRose\ grammar)^{\circ}\end{aligned}$$

As indicated by the type signature, *trees* is a relation. Relations can be thought of as set-valued functions, so given a symbol, *trees* produces the set of all grammatically correct trees with that symbol-type.

The above definition is known as an *unfold*. See Bird and de Moor [5] for a proper discussion of folds and unfolds over relations.

Testing all Grammatically Correct Trees. We now have the two necessary ingredients for the definedness test. We can test an individual tree for definedness and we can generate the set of all grammatically correct trees. We can put them together as follows:

$$\begin{aligned} &circ :: Symbol \leftrightarrow (Inh^{\#} \rightarrow Syn^{\#}) \\ &circ = trans\ ag^{\#} \circ trees \end{aligned}$$

Given a symbol, *circ* produces the test results for all trees of that type. As we explained earlier, these results form a set of finite size, despite the fact that the number of trees involved might be infinite. So the question is: how do we compute the results in a finite amount of time? The answer is provided by the hylomorphism theorem (see the appendix). Recalling that *trans* can be written as a fold and *trees* as an unfold, we see that an equivalent definition for *circ* is:

$$circ = \mu\, \langle c \bullet knit \circ (ag^{\#} \times map\ c) \circ grammar^{\circ} \rangle \tag{4}$$

The result of this fixpoint computation can be computed by *iteration*. Iteration is a technique for computing the value of $\mu\, f$ in finite sized domains. The technique is to compute the sequence $\bot, f(\bot), f(f(\bot)), \ldots$, which is guaranteed to stabilise at $\mu\, f$ within a finite number of steps. In the context of computing *circ*, c is a relation over a finite domain, so we represent it as a set of pairs. We start with the empty set and iterate until c ceases to change.

Testing repmin. Let us evaluate *circ* for repmin. We start with c equal to the empty relation. After one iteration, c' equals:

$$\begin{aligned} c' &= knit \circ (repAG^{\#} \times map\ c) \circ grammar^{\circ} \\ &= \langle\ Tree \mapsto \langle\ a \bullet (True,\ a) \rangle \rangle \end{aligned}$$

After the second iteration:

$$c'' = \langle\ Start \mapsto \langle\ a \bullet (False,\ True) \rangle,\ Tree \mapsto \langle\ a \bullet (True,\ a) \rangle \rangle$$

On the third iteration, we find that $c''' = c''$, so the iteration has stabilised and $circ = c''$. What does this value of *circ* tell us about repmin? On the root node, the *lmin* attribute will never be defined, regardless of whether the inherited *gmin* attribute is defined, but the *ntree* attribute is always defined. On internal nodes of the tree, *lmin* is always defined and *ntree* is defined if *gmin* is defined.

5 Conclusion

We have defined a semantics for attribute grammars based on the lambda calculus and given a new proof of Chirica and Martin's important result [6]. We also derived a definedness test, which is very similar to the traditional circularity test. However, our test also encompasses the closure test. The use of the hylomorphism theorem is very important in our derivation of the definedness test and it results in an algorithm based on iteration. We think this may also help to explain why the algorithm for the circularity test is based on iteration.

Our decision to use rose trees comes with one major drawback: loss of type safety. If we had used algebraic datatypes to define the abstract syntax, then they would automatically be grammatically correct. We would not have needed to discuss "The Grammar" in Section 4.2. It is also common for different grammar symbols to have different attributes. For example, in *repmin* the *Start* symbol does not need a *locmin* or a *gmin* attribute. In our model, it is not possible to distinguish between the symbols in this way. It would be interesting to explore the use of category theory to extend our results to an arbitrary recursive datatype.

Acknowledgements. I am supported by a grant from Microsoft Research. I would like to thank Eric Van Wyk, Ganesh Sittampalam and Oege de Moor for all their help with this paper. I would also like to thank my father for his help with the paper and for teaching me about fixpoints.

References

1. C. J. Aarts. Galois connections presented calculationally. Graduating Dissertation, Department of Computing Science, Eindhoven University of Technology. Available from: `http://www.cs.nott.ac.uk/\char'176rcb/MPC/galois.ps.gz`, 1992.
2. R. C. Backhouse and K. S. Backhouse. Abstract interpretations for free. Available from: `http://www.cs.nott.ac.uk/\char'176rcb/papers/papers.html`.
3. R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.
4. R. S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2 edition, 1998.
5. R. S. Bird and O. de Moor. *Algebra of Programming*, volume 100 of *International Series in Computer Science*. Prentice Hall, 1997.
6. L. M. Chirica and D. F. Martin. An order-algebraic definition of Knuthian semantics. *Mathematical Systems Theory*, 13(1):1–27, 1979.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
8. O. de Moor, K. Backhouse, and S. D. Swierstra. First-class attribute grammars. *Informatica*, 24(3):329–341, 2000.

9. R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *SIGPlan '86 Symposium on Compiler Construction*, pages 85–98, Palo Alto, CA, June 1986. Association for Computing Machinery, SIGPlan.
10. K. Gondow and T. Katayama. Attribute grammars as record calculus — a structure-oriented denotational semantics of attribute grammars by using cardelli's record calculus. *Informatica*, 24(3):287–299, 2000.
11. T. Johnsson. Attribute grammars as a functional programming paradigm. In Gilles Kahn, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 154–173, Portland, OR, September 1987. Springer-Verlag.
12. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–146, 1968.
13. D. E. Knuth. Semantics of context-free languages: Correction. *Mathematical Systems Theory*, 5:95–96, 1971.
14. M. Kuiper and S. D. Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands CSN '87*, 1987.
15. Mathematics of Program Construction Group, Eindhoven University of Technology. Fixed-point calculus. *Information Processing Letters Special Issue on The Calculational Method*, 53:131–136, 1995.
16. S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Series in Computer Science. Prentice Hall, 1987.
17. J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Proceedings 9th IFIP World Computer Congress, Information Processing '83, Paris, France, 19–23 Sept 1983*, pages 513–523. North-Holland, Amsterdam, 1983.
18. M. Rosendahl. Strictness analysis for attribute grammars. In *PLILP'92*, volume 631 of *LNCS*, pages 145–157. Springer-Verlag, 1992.
19. E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. Compiler Construction 2002.
20. P. Wadler. Theorems for free! In *FPCA'89, London, England*, pages 347–359. ACM Press, September 1989.

A Definitions and Theorems

Pairs $\quad fst\ (x, y) = x \qquad snd\ (x, y) = y \qquad (f \times g)\ (x, y) = (f\ x,\ g\ y)$

Sequences. Throughout this paper, we use sequences rather than lists. Our motivation is that this simplifies the use of the abstraction rule (see below). The type of sequences is:

$$\textbf{type}\ Seq\ \alpha\ =\ Nat^{+}\ \rightarrow\ \alpha$$

Every sequence xs in this document has a finite length n, which means that $\forall \langle k > n \bullet\ xs_k = \bot \rangle$. (Hence, sequences should be over a domain with a $\bot$ element.) We sometimes use the notation $[a, b, c]$ to denote a finite length sequence and xs_k is equivalent to $xs\ k$. Some useful operations on sequences are:

$$
\begin{array}{ll}
\mathit{map} & :: (\alpha \to \beta) \to \mathit{Seq}\,\alpha \to \mathit{Seq}\,\beta \\
\mathit{map}\,f\,\mathit{xs} & = f \circ \mathit{xs} \\
\\
\mathit{appSeq} & :: \mathit{Seq}\,(\alpha \to \beta) \to \mathit{Seq}\,\alpha \to \mathit{Seq}\,\beta \\
\mathit{appSeq}\,\mathit{fs}\,\mathit{xs}\,k & = \mathit{fs}\,k\,(\mathit{xs}\,k) \\
\\
\mathit{zipWith} & :: (\alpha \to \beta \to \gamma) \to \mathit{Seq}\,\alpha \to \mathit{Seq}\,\beta \to \mathit{Seq}\,\gamma \\
\mathit{zipWith}\,f\,\mathit{xs}\,\mathit{ys}\,k & = f\,(\mathit{xs}\,k)\,(\mathit{ys}\,k)
\end{array}
$$

Note that *appSeq* and *zipWith* are equivalent to the S and S' combinators, respectively (see Peyton Jones [16, pages 260 and 270]). Therefore, the claim on page 148 can be stated as, for all c_1, c_2, xs, ys and k:

$$S'\ c_1\ xs\ (S'\ c_2\ ys\ zs)\ k\ =\ S\ (S'\ (\circ)\ (c_1 \circ xs)\ (c_2 \circ ys))\ zs\ k$$

The proof is a simple manipulation of the definitions of S and S'.

Rose Trees. Our rose trees differ from Bird's [4, page 195], because we use sequences, rather than lists:

$$\textbf{data}\ \mathit{Rose}\ \alpha\ =\ \mathit{Node}\ \alpha\ (\mathit{Seq}\ (\mathit{Rose}\ \alpha))$$

All our rose trees are finite in size. Some useful functions are:

$$
\begin{array}{ll}
\mathit{split} :: \mathit{Rose}\ \alpha \to (\alpha, \mathit{Forest}\ \alpha) & \mathit{merge} :: (\alpha, \mathit{Forest}\ \alpha) \to \mathit{Rose}\ \alpha \\
\mathit{split}\ (\mathit{Node}\ x\ \mathit{xs}) = (x, \mathit{xs}) & \mathit{merge}\ (x, \mathit{xs}) = \mathit{Node}\ x\ \mathit{xs}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{mapRose} :: (\alpha \to \beta) \to \mathit{Rose}\ \alpha \to \mathit{Rose}\ \beta & \mathit{root} :: \mathit{Rose}\ \alpha \to \alpha \\
\mathit{mapRose}\ f\ (\mathit{Node}\ x\ \mathit{xs}) = & \mathit{root} = \mathit{fst} \circ \mathit{split} \\
\quad \mathit{Node}\ (f\ x)\ (\mathit{map}\ (\mathit{mapRose}\ f)\ \mathit{xs}) &
\end{array}
$$

$$\mathit{appRose} :: \mathit{Rose}\ (\alpha \to \beta) \to \mathit{Rose}\ \alpha \to \mathit{Rose}\ \beta$$
$$\mathit{appRose}\ (\mathit{Node}\ f\ \mathit{fs})\ (\mathit{Node}\ x\ \mathit{xs}) = \mathit{Node}\ (f\ x)\ (\mathit{zipWith}\ \mathit{appRose}\ \mathit{fs}\ \mathit{xs})$$

$$\mathit{foldRose} :: ((\alpha, \mathit{Seq}\ \beta) \to \beta) \to \mathit{Rose}\ \alpha \to \beta$$
$$\mathit{foldRose}\ r\ (\mathit{Node}\ x\ \mathit{xs}) = r\ (x, \mathit{map}\ (\mathit{foldRose}\ r)\ \mathit{xs})$$

The hylomorphism theorem [5, page 142] on rose trees states that, for all r, s:

$$(\mathit{foldrose}\ r) \circ (\mathit{foldrose}\ s)^\circ\ =\ \mu\ \langle h \bullet r \circ (\mathit{id} \times \mathit{map}\ h) \circ s^\circ \rangle$$

Fixpoint Theorems. A proper introduction to these theorems is given by the Eindhoven MPC group [15]. The theorems assume that the fixpoints exist.

1. Rolling Rule. *If f is a monotonic function of type $A \to B$ and g is a monotonic function of type $B \to A$ then:* $\mu\ (f \circ g) = f\ (\mu\ (g \circ f))$.
2. Abstraction Rule. *Suppose f is a function of type $A \to B \to B$. If f is monotonic in both its arguments then:* $\mu\ (S\ f) = \langle x \bullet \mu\ (f\ x) \rangle$. *In the context of sequences, this means that for all xs:* $\mu\ (\mathit{appSeq}\ \mathit{xs}) = \mathit{map}\ \mu\ \mathit{xs}$.

3. Mutual Recursion Rule. *Suppose f is a monotonic function of type $(A, B) \to (A, B)$. If we define $f_1 = fst \circ f$ and $f_2 = snd \circ f$, then:*

$$\begin{array}{rcl} \mu f & = & (\ \mu\,\langle x \bullet f_1\ (x, p\ x)\rangle,\ \mu\,\langle y \bullet f_2\ (q\ y, y)\rangle\ ) \\ & & \textbf{where}\ p\ x\ =\ \mu\,\langle v \bullet f_2\ (x, v)\rangle \\ & & \qquad\quad q\ y\ =\ \mu\,\langle u \bullet f_1\ (u, y)\rangle \end{array}$$