Integrating attribute grammar and functional programming language features

Ted Kaminski and Eric Van Wyk

Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN 55455, USA
tedinski@cs.umn.edu, evw@cs.umn.edu

Abstract. While attribute grammars (AGs) have several features making them advantageous for specifying language processing tools, functional programming languages offer a myriad of features also well-suited for such tasks. Much other work shows the close relationship between these two approaches, often in the form of embedding AGs into lazy functional languages. This paper continues in this tradition, but in the other direction, by integrating various functional language features into AGs. Specifically we integrate rich static types (including parametric polymorphism, typed distinctions between decorated and undecorated trees, type inference, and generalized algebraic data-types) and pattern-matching, all in a manner that maintains familiar and convenient attribute grammar notations and especially their highly extensible nature.

1 Introduction

Attribute grammars[8] are a programming paradigm for the declarative specification of computations over trees, especially of interest in specifying the semantics of software languages. The underlying context free grammar of the language provides the structure for syntax-directed analysis, and synthesized and inherited attributes provide a convenient means for declaratively specifying the flow of information up and down the tree.

Over the years many additions to this formalism have been proposed to increase the expressiveness, flexibility, extensibility, and convenience of attribute grammars. Higher-order attributes [19] were introduced to, among other things, end the hegemony of the original syntax tree. Many computations are easier over transformed syntax trees, which is why compilers often transform between several intermediate languages, or are not possible without dynamically generating arbitrarily larger trees. Through higher-order attributes, these new trees can be constructed, stored in attributes, and also decorated with attributes.

Reference attributes [5] were introduced to handle non-local dependencies across a tree, and are often described as superimposing a graph structure on top of the abstract syntax tree. A typical use of reference attributes is obtaining a direct reference to the declaration node of an identifier at a use site of that identifier. These help dramatically in allowing specifications to be written at a high-level.

Forwarding [16] and production attributes were introduced in order to solve an extensibility problem for attribute grammars. Independently designed *language extensions* can be written as attribute grammar fragments that can add new productions (new language constructs) or new attributes and attribute equations to existing productions (new analysis or translation). But, these extensions may not compose because the new attributes will not have defining equations on the new productions. Forwarding provides a solution to this problem by permitting new productions to *forward* any queries for unspecified attributes to a *semantically equivalent* tree in the host language, where these attributes would have been defined. This tree does not have to be statically determined, and can be computed dynamically, often by using higher-order attributes. Forwarding has also proven useful for more than simply achieving this extensibility property. It is an easy way of de-sugaring, while still being able to explicitly define attributes on the extension syntax where needed. It is also very useful for specifying static dispatch based on, for example, types.

There are many other useful extensions to attribute grammars such as remote attributes and collections [1], circular attributes [2], generic attribute grammars [12], and more. In this paper, however, we will only be considering the three described above.

Functional programming languages also offer a number of compelling features, such as strong static typing, parametric polymorphism, type inference, pattern matching, and generalized algebraic data types. We are interested in integrating these feature into attribute grammars in order to enjoy the best of both worlds. We have a number of goals in doing so:

1. *Safety.* The language should have features that help us identify and prevent bugs in our attribute grammar.
2. *Synergy.* The features should work together and not be separate, disjointed parts of the language.
3. *Simplicity.* It should not be a heavy burden on implementer of the attribute grammar specification language.
4. *Extensible.* We do not want to compromise on one of the biggest advantages attribute grammars and forwarding provide.
5. *Fully-featured.* These features should be integrated with attribute grammars well enough that they are still as useful and powerful as they are in functional languages.
6. *Natural.* Notation should be convenient, sensible and not overly cumbersome, and error messages should be appropriate and clear.

One of the strongest motivating examples for this integration is that of using attribute grammar constructs for representing type information in a language processor. We would definitely like the extensibility properties that attribute grammars possess: we should be able to create new types (new productions), as well as add new attributes to existing types. For example, we might want to extend a language with a specialized list type, and add a new production attribute to all types for handling an append operation that dispatches on type. If the attribute grammar is embedded in Haskell, Java, or similar languages, the

algebraic data types and classes available in these languages cannot meet these requirements to the same satisfaction that attribute grammars can.

We would equally like a number of functional programming language features for this application. Checking for type equality (or unifying two types) without making use of pattern matching often results in programmers creating an `isFoo` attribute for every production `foo`, along with attributes used only to access the children of a production, and using those to test for equality. This is essentially reinventing pattern matching, badly. These tedious "solutions" are elegantly avoided by allowing pattern matching on nonterminals.

Pattern matching proves useful far beyond just types, however. There are many cases in language processing where we care about the *local structure* of syntax trees, and these are all ideal for handling with pattern matching. In many cases, the expressiveness pattern matching provides can be difficult to match with attributes since a large number of attributes are typically necessary to emulate a pattern.

Having decided to represent types using grammars, we may ask what other "data structure"-like aspects of the language definition might benefit from being represented as grammars as well? One example is the type of information typically stored in a symbol table. Extensible *environments*, where new language features can easily add new namespaces, scopes, or other contextual information, become possible simply by representing them as grammars with existing attribute grammar features. New namespaces, for example, can simply be new attributes on the environment nonterminal, and new information can be added to the environment in an extensible way through new productions that make use of forwarding. But without parametric polymorphism, a specialized nonterminal has to be rewritten for every type of information that we wish to store in the environment, which quickly becomes tedious. But with it, we can design environments and symbol table structures in a generic way so that they can be implemented once in a library and reused in different language implementations.

We make the following contributions:

- We describe a small attribute grammar language Ag, a subset of Silver [17], that captures the essence of most attribute grammar specification languages (section 2.)
- We show how types help simplify the treatment of higher-order, reference, and production-valued attributes (section 2.)
- We describe a type system for Ag that carefully integrates all of the desired features (section 3). We also identify and work around a weakness of applying the simple Hindley-Milner type system to attribute grammars (section 3.2.)
- We describe a method for using types to improve the notation of the language, by automatically inferring whether a child identifier intends to reference the originally supplied tree (on which values for attributes have not be computed) or the version of the tree that has been decorated with attributes (section 3.3.)
- We describe a new interaction with forwarding that permits pattern matching to be used on attribute grammars without compromising the extensibility of the grammar (section 4.)

$$
\begin{array}{rcl}
T & ::= & n_v \;\mid\; n_n < \overline{T} > \;\mid\; \texttt{Decorated}\; n_n < \overline{T} > \;\mid\; \texttt{Production} (n_n < \overline{T} > \texttt{::=}\; \overline{T}) \\
D & ::= & \cdot \;\mid\; \texttt{nonterminal}\; n_n < \overline{n_v} > \;;\; D \\
 & \mid & \texttt{synthesized attribute}\; n_a < \overline{n_v} > \texttt{::}\, T \;;\; D \\
 & \mid & \texttt{inherited attribute}\; n_a < \overline{n_v} > \texttt{::}\, T \;;\; D \\
 & \mid & \texttt{attribute}\; n_a < \overline{T} > \;\texttt{occurs on}\; n_n < \overline{n_v} > \;;\; D \\
 & \mid & \texttt{production}\; n \;\; n_l \texttt{::}\, n_n < \overline{T} > \texttt{::=}\; \overline{n \texttt{::}\, T} \;\{\; \overline{S} \;\}\; D \\
S & ::= & n \,.\, n_a \;\texttt{=}\; E \;\;; \quad \mid \texttt{forwards to}\; E \;\{\; \overline{A} \;\} \quad ; \\
A & ::= & n_a \;\texttt{=}\; E \\
E & ::= & n \;\mid\; E_f(\overline{E}) \;\mid\; E \,.\, n_a \;\mid\; \texttt{decorate}\; E\; \texttt{with}\; \{\; \overline{A} \;\} \quad \mid \texttt{new}\; E
\end{array}
$$

Fig. 1. The language AG

2 The AG language

A number of Silver features are omitted from the language AG, as we wish to focus on those parts of the language that are interesting from a typing and semantics perspective and are generally applicable to other attribute grammar languages. To that end, terminals, other components related to parsing and concrete syntax, aspects productions, collection and local attributes, functions, operations on primitive types, as well as many other basic features are all omitted from AG. For most of these features, there are no additional complications in adding them to scale AG up to Silver.

The grammar for the language AG is given in Fig. 1. Names of values (*e.g.* productions and trees) are denoted n, and we follow the convention of denoting nonterminal names as n_n, attribute names as n_a, and type variables as n_v.

A program in AG is a set of declarations, denoted D. These declarations would normally be mutually recursive, but for simplicity of presentation, we consider them in sequence in AG. (Mutual recursion could cause problems for type reconstruction, but we are not relying on type reconstruction in any way for declarations in AG or Silver.) The forms of declaration should be relatively standard for attribute grammars; we take the view of attributes being declared separately from the nonterminals on which they occur.

Note that nonterminals are parameterized by a set of type variables ($\overline{n_v}$). We will adopt the convention of omitting the angle brackets whenever this list is empty. Attributes, too, are parameterized by a set of type variables, and it is the responsibility of the `occurs on` declaration to make clear the association between any variables an attribute is parameterized by, and those variables the nonterminal is parameterized by.

Production declarations give a name (n_l) and type ($n_n < \overline{T} >$) to the nonterminal they construct. The name is used to define synthesized attributes for this production or access inherited attributes given to this production inside the body of the production ($\overline{S}$). Each of the children of the production is also given a name and type, and the body of the production consists of a set of (what we will call) statements (S).

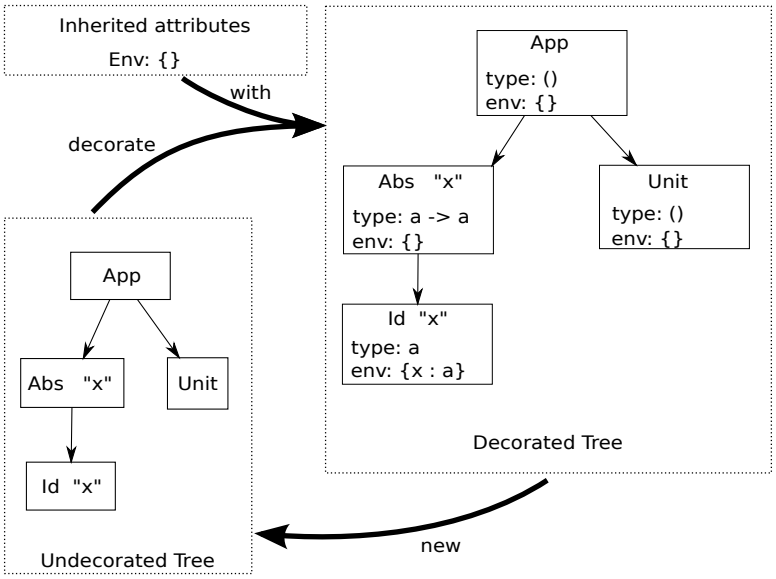

Fig. 2. The distinction between undecorated and decorated trees, and the operations `decorate` and `new` on them. The trees are a representation of the lambda calculus expression $(\lambda x.x)()$.

The attribute definition statement may define synthesized attributes for the node created by the current production, or inherited attributes for its children, depending on whether the name (n) is the left hand side (n_l in production declarations) or the name of a child, respectively.

Forwarding is simply another form of equation that can be written as part of the production body. Forwarding works by *forwarding* requests of any attributes not defined by this production to the *forwarded-to* tree (E). Any inherited attributes requested by the *forwarded-to* tree may be supplied in A, or will otherwise be forwarded to the inherited attributes supplied to this node of the tree. If no new synthesized attribute equations are given in a production, and no inherited attribute equations are given in the forward, then forwarding would behave identically to simple macro expansion.

Expressions are denoted E. Production application (tree construction) and attribute access are standard, but there are two new forms of expression available: `new` creates an undecorated version of a tree from a decorated tree, and `decorate` creates a new decorated tree by supplying it with a list of inherited attribute definitions, denoted A. These operation are illustrated visually in Fig. 2.

In the specification of types, T, we see that every (parameterized) nonterminal n_n produces two distinct, but related, types:

– the undecorated type, denoted n_n, is for trees without computed attribute values (these play the same role as algebraic data types in ML or Haskell)
– the decorated type, denoted `Decorated` n_n, is a tree that is decorated with attributes, created by supplying an undecorated tree with its inherited attributes.

The observed distinction between these two goes back at least as far as [3], and a type distinction between decorated and undecorated trees shows up naturally in functional embeddings of attribute grammars (such as in [6]), but with less familiar notation. Despite this, to the best of our knowledge, the type distinction has not been deliberately exposed as part of an attribute grammar specification language before. For a comparison with other languages, see related work in section 5.

Distinguishing these two kinds of trees by types provides the following advantages:

– *Enhanced Static Type Safety.* Object-oriented embeddings in particular often do not make this distinction, allowing either kind of tree to be used incorrectly. This stems from the ability to set inherited attributes by side-effects on objects representing trees.
– *Simplicity.* Higher-order [19], reference [5] and production-valued [16] attributes are just ordinary attributes of different types (respectively, n_n, `Decorated` n_n, and `Production`(...)), and need no special treatment by the evaluator or the language.
– *Maintain expressiveness.* Productions can deliberately take decorated trees as children, for example, allowing a tree to share sub-trees with other trees. This is technically allowed in other systems with reference attributes, but at the expense of type safety, and it's not clear it is an intended feature.
– *Convenience.* In section 3.3 we show how these types can be used to provide a convenient notation in Ag.

A small example of a grammar for boolean propositions is given in Fig. 3. The example assumes the existence of a primitive boolean type, not included in our definition of Ag, and one shorthand notation: the `with` syntax of the nonterminal declaration stands in for `occurs on` declarations of Ag for each of the attributes that follow it.

The grammar shows the use of a few of the basic features of attribute grammars, in the notation of Ag. The `eval` attribute is an ordinary synthesized attribute, while the `negation` attribute is a so-called higher-order attribute. In Ag, we can see this is just an ordinary attribute with a different type. The `implies` production shows the use of forwarding in a very macro-like fashion, while the `iff` production demonstrates that it's still possible to provide equations for attributes when forwarding.

The example grammar in Fig. 4 shows the use of the polymorphic syntax for nonterminal, attribute, and production declarations for the very simple example of the pair data structure. We will be referring back to these two example grammars later on in the paper to illustrate some subtle details.

```
nonterminal Expr with eval, negation;
synthesized attribute eval :: Boolean;
synthesized attribute negation :: Expr;

production and
e::Expr ::= l::Expr r::Expr
{ e.eval = l.eval && r.eval;
  e.negation = or(not(l),not(r));
}
production or
e::Expr ::= l::Expr r::Expr
{ e.eval = l.eval || r.eval;
  e.negation = and(not(l),not(r));
}
production not
e::Expr ::= s::Expr
{ e.eval = !s.eval;
  e.negation = s;
}
production literal
e::Expr ::= b::Boolean
{ e.eval = b;
  e.negation = literal(!b);
}
production implies
e::Expr ::= l::Expr r::Expr
{ forwards to or(not(l),r);
}
production iff
e::Expr ::= l::Expr r::Expr
{ e.eval = l.eval == r.eval;
  forwards to and(implies(l,r),
                  implies(r,l));
}
```

Fig. 3. An simple example grammar for boolean propositions, written in AG

```
nonterminal Pair<a b>;
synthesized attribute fst<a> :: a;
synthesized attribute snd<a> :: a;
attribute fst<a> occurs on Pair<a b>;
attribute snd<d> occurs on Pair<c d>;

production pair
p::Pair<a b> ::= f::a  s::b
{ p.fst = f;
  p.snd = s;
}
```

Fig. 4. An simple example defining a pair type, written in AG

3 The AG type system

3.1 The type inference rules

Each of the nonterminals in the (meta) language AG has a rather special typing relation, and so we will describe them in some detail.

$$\boxed{N; P; S; I; O; \Gamma \vdash D} \text{ Declarations}$$

Here N, P, S, I, and O represent, respectively, declared nonterminal types, production names, synthesized attributes, inherited attributes, and occurs declarations. These reflect the various components of an attribute grammar specification, but here we specify them explicitly. Since these do not change except at the top level of declarations (D), we will omit writing them for the other typing relations and consider them to be implicitly available.

$$\boxed{L; R; \Gamma \vdash S} \text{ Production body statements}$$

L is the name and type of the left-hand side symbol of the production (the type the production constructs.) R is the set of name/type pairs for the right-hand side symbols (the children) of the production. These are used to distinguish when it is acceptable to defined inherited or synthesized attributes inside the production statements.

$\boxed{X; \Gamma \vdash A}$ Inherited attribute assignments

Here, X is the type of the nonterminal that inherited attributes are being supplied to by `decorate` expressions and `forwards to` statements.

$\boxed{\Gamma \vdash E : T}$ Expressions

This is the standard relation for expressions, except for N, P, S, I, and O that are implicitly supplied.

Inference rules. The type inference rules for AG are shown in Fig. 5. In all rules, we omit explicitly checking the validity of types (T) written in the syntax. All that is required to ensure types are valid is that n_n actually refer to a declared nonterminal, with an appropriate number of parameter types. Wherever n_n appears on its own in the syntax, however, we will write these checks explicitly. Additionally, we always require lists of type variables $\overline{n_v}$ to contain no duplicates.

We use $fv(T)$ to represent the free type variables of a type T. This may also be applied to many types ($\overline{T}$), in which case it is the union of the free type variables. To ensure that different sequences of types or type variables have the same number of elements, we use the notation $\overline{T} \forall k$ to indicate that there are k elements in the sequence $\overline{T}$.

The rule D-NT declaring nonterminals is straightforward and adds the nonterminal type to N. We omit basic checks for redeclarations here for brevity. The rule D-SYN adds the type of the synthesized attribute to S and ensures the type of the attribute is closed under the variables it is parameterized by. The rule for inherited attributes is symmetric and not shown.

The rule D-OCC requires some explanation. The actual value stored in O for an occurrence of an attribute on a nonterminal is a function, α, from the nonterminal's type to the type of the attribute. We write $[\overline{n_v \mapsto T}]$ to represent a substitution that maps each type variable to its respective type. The definition of α looks complex but is quite simple: first, we are interested in the type of the attribute (T_a), so that is what the substitution is applied to. We want to equate the variables declared as parameters of the nonterminal (n_{vdn}) with both the variables written in this occurs declaration (n_v) and with the types supplied as a parameter to this function (T_p). Finally, we want to equate the type variables that are parameters of the attribute (n_{vda}) with the actual type supplied for those parameters in this occurs declaration (T). The directions of these rewrites are simply such that no type variables from these declarations "escape" into the

$$\frac{N \cup n_n < \overline{n_v} >; P; S; I; O; \Gamma \vdash D}{N; P; S; I; O; \Gamma \vdash \texttt{nonterminal}\ \ n_n < \overline{n_v} >\ \ ;\ \ D}\ \text{(D-NT)}$$

$$\frac{fv(T) \setminus \overline{n_v} = \emptyset \qquad N; P; S \cup n_a < \overline{n_v} > : T; I; O; \Gamma \vdash D}{N; P; S; I; O; \Gamma \vdash \texttt{synthesized attribute}\ \ n_a < \overline{n_v} > \texttt{::}\ T\ \ ;\ \ D}\ \text{(D-SYN)}$$

$$\frac{\begin{array}{c} fv(\overline{T}) \setminus \overline{n_v} = \emptyset \quad n_a < \overline{n_{vda} \forall k} > : T_a \in S \cup I \qquad n_n < \overline{n_{vdn} \forall j} > \in N \\ \alpha(n_n < \overline{T_p \forall j} >) = (\overline{[n_{vda} \mapsto T]} \circ \overline{[n_v \mapsto n_{vdn}]} \circ \overline{[n_{vdn} \mapsto T_p]})(T_a) \\ N; P; S; I; O \cup n_a @ n_n = \alpha; \Gamma \vdash D \end{array}}{N; P; S; I; O; \Gamma \vdash \texttt{attribute}\ \ n_a < \overline{T \forall k} >\ \ \texttt{occurs on}\ \ n_n < \overline{n_v \forall j} >\ \ ;\ \ D}\ \text{(D-OCC)}$$

$$\frac{\begin{array}{c} n_n < \overline{n_{vdn} \forall k} > \in N \\ n_l : n_n < \overline{T_n} >; \overline{n_c : T_c}; \Gamma \cup n_l : \texttt{Decorated}\ n_n < \overline{T_n} > \cup \overline{n_c : dec(T_c)} \vdash \overline{S} \\ T_p = \texttt{Production}\,(n_n < \overline{T_n} > \texttt{::=}\ \overline{T_c}) \\ N; P \cup n; S; I; O; \Gamma \cup n : \forall fv(T_p).T_p \vdash D \end{array}}{N; P; S; I; O; \Gamma \vdash \texttt{production}\ \ n\ \ \ n_l :: n_n < \overline{T_n \forall k} > \texttt{::=}\ \ \overline{n_c :: T_c}\ \ \{\ \ \overline{S}\ \ \}\ \ D}\ \text{(D-PROD)}$$

$$\frac{n : T = L, \quad n_a \in S, \quad T = n_n < \overline{T_n} >, \quad n_a @ n_n = \alpha \in O, \quad \Gamma \vdash E : \alpha(T)}{L; R; \Gamma \vdash n\ .\ n_a\ \ \texttt{=}\ \ E\ \ ;}\ \text{(S-SYN)}$$

$$\frac{n : T \in R, \quad n_a \in I, \quad T = n_n < \overline{T_n} >, \quad n_a @ n_n = \alpha \in O, \quad \Gamma \vdash E : \alpha(T)}{L; R; \Gamma \vdash n\ .\ n_a\ \ \texttt{=}\ \ E\ \ ;}\ \text{(S-INH)}$$

$$\frac{n : T = L \qquad \Gamma \vdash E : T \qquad \overline{T; \Gamma \vdash A}}{L; R; \Gamma \vdash \texttt{forwards to}\ \ E\ \ \{\ \ \overline{A}\ \ \}\ \ ;}\ \text{(S-FWD)}$$

$$\frac{n_a \in I \qquad X = n_n < \overline{T_n} > \qquad n_a @ n_n = \alpha \in O \qquad \Gamma \vdash E : \alpha(T)}{X; \Gamma \vdash n_a\ \ \texttt{=}\ \ E}\ \text{(A-INH)}$$

$$\frac{n : \forall \overline{n_v}.T_q \in \Gamma}{\Gamma \vdash n : \overline{[n_v \mapsto \nu]}T}\ \text{(E-VAR)} \qquad \frac{\Gamma \vdash E : \texttt{Decorated}\ n_n < T_n >}{\Gamma \vdash \texttt{new}\ \ E : n_n < T_n >}\ \text{(E-NEW)}$$

$$\frac{\Gamma \vdash E_f : \texttt{Production}\,(T\ \texttt{::=}\ \overline{T_c}) \qquad \overline{\Gamma \vdash E : T_c}}{\Gamma \vdash E_f(\overline{E}) : T}\ \text{(E-APP)}$$

$$\frac{\Gamma \vdash E : \texttt{Decorated}\ n_n < T_n > \qquad n_a @ n_n = \alpha \in O}{\Gamma \vdash E\ .\ n_a : \alpha(n_n < T_n >)}\ \text{(E-ACC)}$$

$$\frac{\Gamma \vdash E : n_n < T_n > \qquad \overline{n_n < T_n >; \Gamma \vdash A}}{\Gamma \vdash \texttt{decorate}\ \ E\ \ \texttt{with}\ \ \{\ \ \overline{A}\ \ \}\ \ : \texttt{Decorated}\ n_n < T_n >}\ \text{(E-DEC)}$$

Fig. 5. Type inference rules AG.

.

resulting type. For example, the function α for `fst` attribute on `Pair` shown in Fig. 4 would map `Pair` $< T\ S >$ to T.

The rule D-PROD has a few very particular details. The types written in the production signature are those types that should be supplied when the production is applied. Inside the body of the production, however, the children and left-hand side should appear decorated. So, if a child is declared as having type `Expr` (as in many of the productions of Fig. 3), then inside the production body, its type is seen to be `Decorated Expr`. To accomplish this, we apply *dec* to the types of the children when adding them to the environment (Γ). *dec*'s behavior is simple: it is the identity function, except that nonterminal types $n_n < \overline{T} >$ become their associated decorated types `Decorated` $n_n < \overline{T} >$. The purpose of this is to reflect what the production does: there will be rules inside the production body ($\overline{S}$) that define inherited attributes for its children, and therefore, the children are being automatically decorated by the production and should be seen as decorated within the production body.

Note that when D-PROD checks the validity of its statements $\overline{S}$, it supplies R with types unchanged (that is, without *dec* applied.) This is also important, as inherited attributes can only be supplied to previously undecorated children. Children of already decorated type already have their inherited attributes, and so this information (R, without *dec* applied) is necessary to distinguish between children that were initially undecorated and those that were already decorated and cannot be supplied new inherited attributes.

The rules S-SYN and S-INH are again symmetric, and apply to the same syntax. Which rule is used depends on whether an inherited attribute is being supplied to a child, or a synthesized attribute is being defined for the production. Note that we use the shorthand $n_a \in S$ simply to mean that it is a declared attribute of the appropriate kind, as we no longer care about the type declared for the attribute specifically, that will be obtained from the occurs declaration via the function α. The only major detail for rule S-FWD is the the expression type is undecorated. Rule A-INH is similar to S-INH except that we obtain the type from the context, rather than by looking up a name.

The rules E-VAR, and E-APP are slight adaptations of the standard versions of these for the lambda calculus. Notice in the rule E-ACC that the expression type is required to be decorated (attributes cannot be accessed from trees that have not yet been decorated with attributes.) In section 3.2, we consider the problem with this rule, as written, where we must know the type of the left hand side in order to report any type at all for the whole expression, due to the function α. The rules E-DEC and E-NEW should look straightforward, based on their descriptions in the previous section, and the visual in Fig. 2.

Generalized algebraic data types A full description of GADTs can be found in [11]. Examples of the utility of GADTs are omitted here for space reasons, but many of the examples in the cited functional programming literature make use of them for syntax trees—the application to attribute grammars should be obvious.

The language AG (as it currently stands, without pattern matching) supports GADTs effortlessly. The type system presented in Fig. 5 needs no changes at all, whether GADTs are allowed or not. The only difference is actually syntactic. If the type of the nonterminal on the left hand side of `production` declarations permits types ($\overline{T}$) inside the angle brackets (as they do in Fig. 1), GADTs are supported. If instead, these are restricted to type variables ($\overline{n_v}$), then GADTs are disallowed. All of the complication in supporting GADTs appears to lie in pattern matching, as we will see in section 4.2 when we introduce pattern matching to AG.

3.2 Polymorphic attribute access problem

We encountered an issue in adapting a Hindley-Milner style type system to attributes grammars. Typing the attribute access expression `e.a` immediately runs into two problems with the standard inference algorithm:

- There is no type we can unify `e` with. The constraint we wish to express is that, whatever `e`'s type, the attribute `a` occurs on it. That is, $n_a@n_n = \alpha$ has to be in O.
- There is no type that we can report as the type of the whole expression, without knowing `e`'s type, because without that nonterminal type we cannot find the function α needed to report the attribute's type.

These problems can occur even in the simplest case of parameterized attributes. For example, we cannot know that `e.fst` means that `e` should be a `Pair` or a `Triple`. Any access of `fst` simply requires knowing what type we're accessing `fst` on.

Fortunately, a sufficient level of type annotations guarantees that we're able to infer the type of the subexpression before needing to report a type for the attribute access expression which is necessary to continue type inference. AG actually requires type annotations for every name introduced into the environment, which results in nearly all subexpressions evaluating to a closed type. For AG this requirement is not a burden, since production declarations and attribute declarations really should provide explicit types (even if we had the option not to), and these are the only value declarations in the language.

Type inference does still provide a significant advantage since it infers the "type parameters" to parameterized productions. In a prototype implementation of parametric polymorphism in Silver [4] without inference one needed to specify, for example, the type of elements in an empty list literal. Explicitly specifying such type parameters quickly becomes tedious.

The major downside of needing type annotations would be for features not present in AG. Lambda functions, let expressions, "local attributes" and so forth where we might like type inference must now all have type annotations, instead.

3.3 Putting types to work

In rule D-PROD, nonterminal children are added to the environment in their decorated form for the body of the production (using the function *dec.*) While this is

correct behavior, it can be inconvenient, as it can lead to a tedious proliferation of `new` wherever the undecorated form of a child is needed, instead.

What we'd like is to have these names refer to *either* of their decorated or undecorated values, and simply disambiguate based upon type. The example grammar in Fig. 3 is already relying on this desired behavior. In the `or` production, we happily access the `eval` attribute from the child `l`, when defining the equation for `eval` on this production. But, we also happily apply the `not` production to `l`, when defining `negation`. As currently written, the type rules would require us to write `new(l)` in the latter case, because the `not` production expects an undecorated value, and `l` is seen as decorated within the production.

The simplest change to the type rules to reflect this idea would be to add a new rule for expressions, able to refer implicitly to the R and L contextual information given to statements:

$$\frac{n : T \in R \cup L}{\Gamma \vdash n : T} \text{ (E-AsIs)}$$

Unfortunately, simply introducing this rule leads to nondeterminism when typing checking. With it, there is no obvious way to decide whether to use it or E-VAR, which is problematic.

To resolve this issue, we introduce a new pseudo-union type of both the decorated and undecorated versions of a nonterminal. But this type, called *Und* for undecorable, will also carry with it a type variable that is specialized to the appropriate decorated or undecorated type when the it is used in one way or the other. This restriction reflects the fact that we need to choose between one of these values or the other.

Und is introduced by altering the *dec* function used in D-PROD to turn undecorated child types into undecorable types, rather than decorated types. An undecorable type will freely unify with its corresponding decorated and undecorated type, but in doing so, refines its corresponding hidden type variable.

$$\begin{array}{ll}
\mathcal{U}(Und\langle n_n < \overline{n_v} >, a\rangle, & n_n < \overline{n_v} >) \texttt{ :- } \mathcal{U}(a, n_n < \overline{n_v} >) \\
\mathcal{U}(Und\langle n_n < \overline{n_v} >, a\rangle, & \texttt{Decorated } n_n < \overline{n_v} >) \texttt{ :- } \mathcal{U}(a, \texttt{Decorated } n_n < \overline{n_v} >) \\
\mathcal{U}(Und\langle n_n < \overline{n_v} >, a\rangle, & Und\langle n_n < \overline{n_v} >, b\rangle) \texttt{ :- } \mathcal{U}(a, b)
\end{array}$$

Now, suppose we have the admittedly contrived example below, with the types of `foo` and `bar` as shown, and we attempt to type the expression invoking `foo`

```
bar :: Production(Baz ::= Expr)
foo :: Production(Baz ::= a Production(Baz ::= a) a)
foo(child1, bar, child2)
```

`child1` will report type $Und\langle Expr, a\rangle$, and `child2` will report $Und\langle Expr, b\rangle$. We will then enforce two constraints while checking the application of `foo`:

$$E ::= \texttt{case}\ \ E\ \ \texttt{of}\ \ \overline{p \to E_p}$$
$$p ::= n_p(\overline{n}) \quad | \quad \_$$

Fig. 6. The pattern extension to AG

$Und\langle Expr, a\rangle = Expr$, which using the first rule above will result in requiring $a = Expr$, then $Und\langle Expr, Expr\rangle = Und(\langle Expr\rangle, b)$ which will using the third rule requires $b = Expr$.

The introduction of this undecorable type is something of a special-purpose hack, but the notation gains are worth it. The notational gains could also be achieved with more sophisticated type machinery (like type classes), but it also seems worthwhile to stick to the simple Hindley-Milner style of type systems.

4 Pattern matching

In this section we consider the extensions that must be made to include pattern matching in AG. The main challenge lies in the interaction of forwarding and pattern matching.

4.1 Adding pattern matching to AG

Fig. 6 shows the extension to expression syntax for patterns. Note that to simplify our discussion, we are considering only single-value, non-nested patterns. Support for nested patterns that match on multiple values at once can be obtained simply by applying a standard pattern matching compiler, such as [20].

As already noted in the introduction, pattern matching can be emulated with attributes, but that emulation comes at the cost of potentially needing many attributes. One possible translation of pattern matching to attributes begins by creating a new synthesized attribute for each match expression, occurring on the nonterminal it matches on, with the corresponding pattern expression as the attribute equation for each production[1]. This also requires every name referenced in that equation to be turned into an inherited attribute that is passed into that nonterminal by the production performing the match. These names not only include children of the production, but also any pattern variables bound by enclosing pattern matching expressions, such as those created by the pattern compiler from multi-value, nested patterns.

This translation actually does not quite work in AG: pattern matching on reference attributes is problematic because they're already decorated values that we cannot supply with more inherited attributes. In practice, though, there are other language features available that can be used to avoid this problem. Still, this is

[1] The observant reader may note here that we have left out wildcards. This is deliberate, and will be considered shortly.

```
nonterminal Type with eq, eqto;
synthesized attribute eq :: Boolean;
inherited attribute eqto :: Type;

abstract production pair
t::Type ::= l::Type r::Type
{ t.eq =
   case t.eqto of
    pair(a, b) ->
      (decorate l with { eqto = a }).eq &&
      (decorate r with { eqto = b }).eq
   | _ -> false
   end;
}
```

```
abstract production tuple
t::Type ::= ts::[Type]
{ forwards to
   case ts of
     [] -> unit()
   | a:[] -> a
   | a:b:[] -> pair(a, b)
   | f:r -> pair(f, tuple(r))
   end;
}
```

Fig. 7. A use of pattern matching in types.

not a good approach for implementing pattern matching. The most prolific data structures are probably also those pattern matched upon the most, and unless the attribute grammar implementation is specifically designed around solving this problem, there will be overhead for every attribute. A `List` nonterminal, for example, could easily balloon to very many attributes that are the result of translated-away patterns, and there could easily be very many more *cons* nodes in memory. The result would not be memory efficient, to say the least.

The true value of considering this translation to attributes is in trying to resolve the problem pattern matching raises for extensibility. Patterns are explicit lists of productions (constructors), something that works just fine for data types in functional languages because data types are closed: no new constructors can be introduced. Nonterminals are not closed, and this is a major friction in integrating these two language features. However, if pattern matching has a successful reduction to attributes, that problem is already solved: forwarding gives us the solution.

But, the translation to attributes is not quite fully specified: what do we do in the case of wild cards? We choose the smallest possible answer that could still allow us to be sure we cover all cases: wild cards will apply to all productions, not already elsewhere in the list of patterns, that *do not forward*. Thus, we would continue to follow forwards down the chain until either we reach a case in the pattern matching expression, or we reach a non-forwarding, non-matching production, in which case we use the wild card. The dual wild card behavior (applying to all productions, not just those that do not forward) would mean that the "look through forwards" behavior of pattern matching would only occur for patterns without wild cards, which seems unnecessarily limiting.

In Fig. 7, we show a very simple example of the use of pattern matching in determining equality of types. (We again take a few small liberties in notation; new in this example is the use of a list type and some notations for it borrowed

from Haskell.) The advantage of interacting pattern matching and forwarding quickly becomes apparent in the example of a tuple extension to the language of types. The tuple type is able to "inherit" its equality checking behavior from whatever type it forwards to, as is normal for forwarding. But, this alone is not sufficient: consider checking two tuples (`tuple([`S, T, U`])`) against each other. The first will forward to `pair(`S`, pair(`T`,` U`))`, and pattern match on the second. But, without the "look-through" behavior we describe here, the pattern will fail to match, as it will look like a `tuple`. With the behavior, it will successfully match the `pair` production it forwards to, and proceed from there.

Alternative wild card behavior. One alternative might be to take advantage of higher level organizational information (not considered in AG) to decide which productions to apply the wildcard case to. For example, the wildcard could apply to all *known* productions wherever the case expression appears (based on imports or host/extension information), instead of all non-forwarding productions. This has the advantage that we wouldn't need to repeat the wildcard case for some forwarding productions in those cases where we'd like to distinguish between a production and the production it forwards to, but it has a few disadvantages as well:

- We may now need to repeat case alternatives if we *don't* want to distinguish between a forward we know about (e.g. for syntactic sugar.)
- The meaning of a case expression might change based on where it appears or by changing the imports of the grammar it exists in.
- A "useless imports" analysis would have to become more complex to ensure no pattern matching expressions would change behavior, as the wildcard of a case expression may be *implicitly* referencing that grammar.

As a result, this behavior has enough additional implementation and conceptual complexities that we have not adopted it.

4.2 Typing pattern matching expressions

Matching on undecorated trees seems to introduce no new interesting behavior different from pattern matching on ordinary data types, which makes sense because in a very real sense undecorated trees are not different from ordinary data types. Pattern matching on decorated trees, however, introduces a couple of interesting behaviors:

- As we saw in the previous section, we can evaluate the forward of a production and allow pattern matching to "look through" to the forward.
- We can also allow pattern variables to extract the decorated children of a production, rather than just the undecorated children.

Further, restricting pattern matching to only apply to decorated trees doesn't lose us anything: if it makes sense to pattern match on an undecorated tree, then the `case` construct can simply decorate a tree with no inherited attributes

$$\frac{\Gamma \vdash E : \texttt{Decorated}\ n_n < \overline{T} > \qquad \Gamma \vdash \overline{p \to E_p} : n_n < \overline{T} > \to T}{\Gamma \vdash \texttt{case}\ \ E\ \ \texttt{of}\ \ \overline{p \to E_p} : T}\ \text{(E-CASE)}$$

$$\frac{\begin{array}{c} n_p \in P \qquad \Gamma \vdash n_p : \texttt{Production}(T_n\ \texttt{::=}\ \overline{T_c}) \\ \theta \in mgu(T_s = T_n) \qquad \theta(\Gamma, \overline{n : dec(T_c)}) \vdash E_p : \theta(T_r) \end{array}}{\Gamma \vdash n_p(\overline{n}) \to E_p : T_s \to T_r}\ \text{(P-PROD)}$$

Fig. 8. The additional typing rules for pattern matching expressions.

to pattern match upon it. As a result, we have decided to just consider pattern matching on decorated trees in AG.

The type rules for patterns are shown in Fig. 8. Notice in E-CASE that the scrutinee expression (E) must be a decorated type. Also note that *dec* is applied directly to T_c in P-PROD. The reason for this is to allow pattern matching to extract the decorated trees corresponding to a node's children. This function (*dec*) must be applied prior to any type information outside the original declaration of the production being considered, in order to be accurate about which children are available as decorated trees. For example, the `pair` production in Fig. 4 would not be decorating its children, even though they might turn out to be a (undecorated) nonterminal type (i.e. a pair of `Expr`), because to the pair production, the types of its children are type variables. Applying *dec* early means that here we see the type of the children as type variables, rather than a specific type, just as the original production would have.

The use of θ in the type rule P-PROD is the cost that we must pay for supporting GADTs in patterns. The details for handling GADTs in patterns are adapted from [11], as this approach seemed especially simple to implement. In that paper, much attention is paid to a notion of *wobbly* and *rigid* types. Thanks to the concessions in type reconstruction we must make due to the attribute access problem discussed in section 3.2, all bindings in AG can be considered rigid in their sense, vastly simplifying the system even more.

The essential idea is to compute a *most general unifier* (θ) between the pattern scrutinee's type and the result type of the production [2]. We then check the right hand side of the alternative, under the assumptions of the unifier. In effect, all this rule is really stating is that whatever type information we learn from successfully matching a particular GADT-like production stays confined to that branch of the pattern matching expression.

[2] The need to concern ourselves with "fresh" most general unifiers in the sense of the cited paper is eliminated again due to the lack of "wobbly" types.

4.3 Other concerns

No new special cases need to be introduced to perform a well-definedness test in the presence of pattern matching, as pattern matching can be translated to attributes (the troubles mentioned earlier are eliminated if we are allowed full-program information), and forwarding can also be translated away to higher order attribute grammars [16].

The standard techniques apply for ensuring exhaustive matching of patterns, except that we only need to consider productions that do not forward as the essential cases to cover.

Although it is often glossed over in descriptions of type systems, it's worth noting that we're allowing type variables to appear in productions' right hand sides (that is, in the children) that do not appear in the left hand side. This corresponds to a notion of existential types in functional languages, but we do not require any special `forall` notation to include them. Background discussion on existential types can be found in [9].

5 Related work

The integration of pattern matching and forwarding we present in this paper is novel. Some aspects of the rest of the system can be found in scattered in other attribute grammar languages in various forms, but not in ways that provide both the type safety and the familiar and convenient notations that we provide here.

In JastAdd [5] and Kiama[14], trees are represented as objects and attribute evaluation mutates the tree effectfully (either directly as in JastAdd or indirectly via memoization as in Kiama.) As a result, both of these languages lack a type distinction between the two kinds of trees. Instead, the user must remember to invoke a special copy method, analogous to our `new` expression, wherever a new undecorated tree is needed. These copy methods do not change the type of the tree, as our `new` operation does, resulting in a lack of the type safety that we have here. UUAG[15] does not appear to support reference attributes, and so the type distinction is irrelevant. In functional embeddings these type distinctions occur naturally but at the notational cost of typically having different names for the two views of the tree and needing to explicitly create the decorated tree from the undecorated one. AspectAG [18] is a sophisticated embedding into Haskell that naturally maintains the type safety we seek but at some loss of notational convenience. It also requires a fair amount of so-called "type-level" programming that is less direct than the Silver specifications, and the error messages generated can be opaque.

Kiama and UUAG, by virtue of their embedding in functional languages, do support parameterized nonterminals and attributes. UUAG side-steps the attribute access problem of section 3.2 by simply not having reference attributes. All attribute access are therefore only on children, which have an explicit type signature provided. UUAG does not appear to support GADT-like productions, but we suspect it could be easily extended to. Both also support pattern matching on nonterminals. In UUAG, this is only supported for undecorated trees,

and its behavior is identical to ordinary pattern matching in Haskell. In Kiama, pattern matching can extract decorated children from a production. But in both cases, use of pattern matching would compromise the extensibility of the specification. Rascal's [7] allows trees to be dynamically annotated with values, similar to adding attribute *occurs-on* specifications dynamically. However, the presence of annotations is not part of the static type system and thus neither is the distinction between decorated and undecorated trees.

Scala's [10] support for pattern matching and inheritance presents the same type of extensibility problem we faced when integrating pattern matching with forwarding. Their solution is a notion of *sealed* classes that simply prevent new classes from outside the current file from inheriting from it directly.

In [13], a type system with constraints powerful enough to capture the α functions created by our *occurs* declarations is presented. To regain full type inference, we believe the basic Hindley-Milner style system must be abandoned in favor of something at least this powerful.

6 Future work

Typing attribute grammars offers a wealth of future work possibilities. The language Ag is not quite suitable for proving soundness results, as writing down operational semantics for it would be overly complicated. Instead, we would like to develop a smaller core *attribute calculus*, with an appropriate operational semantics and obtain a soundness result from that. To get the simple Hindley-Milner type system to apply, we sacrificed the ability to remove some type annotations from the languages. We believe a more powerful core type system (such as [13]) will permit inference to work freely. The type system currently does not permit many forms of functions over data structures to be recast as attributes. For example, quantifiers are not permitted in the right places to allow `if-then-else` to somehow be written as attributes on a `Boolean` nonterminal. Attributes also cannot occur on only some specializations of a nonterminal, which means natural functions like `sum` over a list of integers cannot be recast as attributes. (Such computations are realized as functions in Silver.)

Furthermore, the traditional well-definedness tests for attribute grammars may have another useful interpretation in terms of types, perhaps refining our blunt distinction between undecorated and decorated types. There may also be refinements possible due to the presence of pattern matching. Generic attribute grammars[12] are partly covered by polymorphic nonterminals, except for their ability to describe constraints on the kinds of types that can be incorporated. For example, in Silver, permitting nonterminals to require type variables to be concrete types, permitting these type variables to appear in concrete syntax, and reifying the result before it is sent to the parser generator would be a fantastic addition to the language. Finally, we would like to account for circular attributes, which are extremely useful for fixed point computations. It would be interesting to see if there is a type-based distinction for circular attributes, just as we show for reference, higher-order, and production attributes in this paper.

7 Conclusion

In this paper we have claimed that certain features found in modern functional languages can be added to an attribute grammar specification language to provide a number of benefits. By using types to distinguish decorated and undecorated trees the type system can prevent certain errors and help to provide more convenient notations. Pattern matching on decorated trees adds a measure of convenience and expressiveness (in the informal sense) to attribute grammar specification languages, and crucially, it can be done while maintaining the extensibility possible with forwarding. Parameterized nonterminals and productions can play the same role as algebraic data types in functional languages; they can be used as syntax trees or for more general purpose computations. Furthermore, GADT-like productions are a very natural fit for attribute grammars.

However, in scaling Ag up to Silver, the type annotations requirement to get around the attribute access problem stands in the way of meeting our *full-featured* goal. This means that functions and local attributes, for example, must specify their types.

In integrating these features into an attribute grammar specification language we found that some small modifications to the implementation of Hindley-Milner typing were needed. To meet our goals of having natural and familiar notations (for attribute access and in order to infer if the decorated or undecorated version of a tree is to be used) it was helpful to have direct control over the type system to make modifications so that attribute grammar-specific concerns could be addressed. Supporting GADT-like productions and pattern matching that is compatible with forwarding required similar levels of control of the languages implementation and translation.

We previously added polymorphic lists and a notion of pattern-matching that was not compatible with forwarding using language extensions [17]. While this approach does allow expressive new features to be added to the language, it could not accomplish all of our goals, as adding a new typing infrastructure (for type inference) *replaces* and does not extend the previous type system in Silver. Adding these kinds of features by embedding attribute grammars in a function language or writing a preprocessor that is closely tied to the underlying implementation language can also make it more difficult to achieve these goals. However, an advantage of these approaches that should not be overlooked is that many useful features of the underlying language can be used "for free" with no real effort on the attribute grammar system designer to include them into their system. It is difficult to draw any conclusions beyond noting that these are the sort of trade-offs that AG system implementers, specifically (and DSL implementers, more generally) need to consider.

References

1. Boyland, J.T.: Remote attribute grammars. J. ACM 52(4), 627–687 (2005)
2. Farrow, R.: Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. ACM SIGPLAN Notices 21(7) (1986)

3. Ganzinger, H., Giegerich, R.: Attribute coupled grammars. SIGPLAN Notices 19, 157–170 (1984)
4. Gao, J.: An Extensible Modeling Language Framework via Attribute Grammars. Ph.D. thesis, University of Minnesota, Department of Computer Science and Engineering, Minneapolis, Minnesota, USA (2007)
5. Hedin, G.: Reference attribute grammars. Informatica 24(3), 301–317 (2000)
6. Johnsson, T.: Attribute grammars as a functional programming paradigm. In: Kahn, G. (ed.) Functional Programming Languages and Computer Architecture. LNCS, vol. 274, pp. 154–173. Springer-Verlag (1987)
7. Klint, P., van der Storm, T., Vinju, J.: Rascal: a domain specific language for source code analysis and manipulation. In: Proc. of Source Code Analysis and Manipulation (SCAM 2009) (2009)
8. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2(2), 127–145 (1968), corrections in **5**(1971) pp. 95–96
9. Läufer, K., Odersky, M.: Polymorphic type inference and abstract data types. ACM Trans. on Prog. Lang. and Systems (TOPLAS) 16(5), 1411–1430 (1994)
10. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima, second edn. (2010)
11. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming. pp. 50–61. ACM (2006)
12. Saraiva, J., Swierstra, D.: Generic Attribute Grammars. In: 2nd Workshop on Attribute Grammars and their Applications. pp. 185–204 (1999)
13. Schrijvers, T., Peyton Jones, S., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for GADTs. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. pp. 341–352. ACM (2009)
14. Sloane, A., Kats, L., Visser, E.: A pure object-oriented embedding of attribute grammars. In: Proc. of Language Descriptions, Tools, and Applications (LDTA 2009). ENTCS, vol. 253, pp. 205–219. Elsevier Science (2010)
15. Swierstra, S., Alcocer, P., Saraiva, J.: Designing and implementing combinator languages. In: Proc. Third International Summer School on Advanced Functional Programming. LNCS, vol. 1608, pp. 150–206. Springer (1999)
16. Van Wyk, E., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in attribute grammars for modular language design. In: Proc. 11th Intl. Conf. on Compiler Construction. LNCS, vol. 2304, pp. 128–142 (2002)
17. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. Science of Computer Programming 75(1–2), 39–54 (January 2010)
18. Viera, M., Swierstra, S.D., Swierstra, W.: Attribute grammars fly first-class: How to do aspect oriented programming in haskell. In: Proc. of 2009 International Conference on Functional Programming (ICFP'09) (2009)
19. Vogt, H., Swierstra, S.D., Kuiper, M.F.: Higher-order attribute grammars. In: ACM Conf. on Prog. Lang. Design and Implementation (PLDI). pp. 131–145 (1990)
20. Wadler, P.: Efficient compilation of pattern matching. In: The Implementation of Functional Programming Languages, pp. 78–103. Prentice-Hall (1987)