Modular well-definedness analysis for attribute grammars[⋆]

Ted Kaminski and Eric Van Wyk

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

Abstract. We present a modular well-definedness analysis for attribute grammars. The global properties of completeness and non-circularity are ensured with checks on grammar modules that require only additional information from their dependencies. Local checks to ensure global properties are crucial for specifying extensible languages. They allow independent developers of language extensions to verify that their extension, when combined with other independently developed and similarly verified extensions to a specified host language, will result in a composed grammar that is well-defined. Thus, the composition of the host language and user-selected extensions can safely be performed by someone with no expertise in language design and implementation. The analysis is necessarily conservative and imposes some restrictions on the grammar. We argue that the analysis is practical and the restrictions are natural and not burdensome by applying it to the Silver specifications of Silver, our boot-strapped extensible attribute grammar system.

1   Introduction

There has been considerable interest in extensible language frameworks and mechanisms for defining language implementations in a highly modular manner. Many of these frameworks allow *language extensions* that add new syntax and new semantic analyses to be composed with the implementation of a so-called *host language*, such as C or Java, resulting in an extended language with new, possibly domain-specific, features. It is our contention that for these frameworks to be useful to programmers and more widely used, the language extensions need to be able to be developed independently and the process of composing the host language with a selected set of extensions needs to be easily doable by programmers with no knowledge of language design and implementation.

There are several systems that support, to varying degrees, the development of modular and composable language extensions. For example, JastAdd [6] is a system based on attribute grammars (AGs) extended with reference attributes [8] which has been used to develop extensible compilers for Java [5] and Modelica. MetaBorg [4] is based on term rewriting systems with strategies [21] and has been

---

used to build extensible specifications of Java and other languages. SugarJ [7] is an extension to Java in which new syntax to SugarJ can be defined in imported libraries written in SugarJ. Our system, Silver [17, 9], is an attribute grammar system with *forwarding* [16], a mechanism for language extension that is useful for combining extensions developed independently, which has also been used to define an extensible specification of Java [18] and other languages.

Our work is motivated by two important questions related to how easy it is for a non-expert to use these frameworks to create new languages from independently developed language extensions.

1. *How easy is it* to compose the host and extension specifications? Must significant glue code be written to combine them, or can one basically direct the tools to compose the host and selected extensions automatically?
2. *What assurances are provided* to the user that the selected extensions will in fact be composable such that the composition defines a working compiler, translator, or interpreter for the specified extended language?

The formalisms on which the above systems are based (context free grammars (CFG), attribute grammars, and term rewriting systems) are all quite easily and naturally composed, and thus adequately address our first question above. The problem arises with the second question: the composition may not have desirable or required properties for that formalism. For example, CFGs compose but the resulting grammar may be ambiguous and thus not suited as a specification for a parser. Similarly, composed AGs may not be complete (i.e. missing required attribute equations) or it may contain circularities in the attribute definitions.

In 2009, Schwerdfeger and Van Wyk described a *modular determinism analysis* for context free grammar fragments that could be run by the language extension developer to verify that their extension CFG, when composed with the host language CFG and other independently developed and similarly verified extension CFGs, would result in a deterministic grammar, from which a conflict-free LR(1) parse table could be constructed [14]. Formally, this was expressed as

$$(\forall i \in [1, n].isComposable(CFG^H, CFG_i^E) \land \;\; conflictFree(CFG^H \cup \{CFG_i^E\}))$$
$$\implies conflictFree(CFG^H \cup \{CFG_1^E, \dots, CFG_n^E\})$$

Note that each extension grammar CFG_i^E is tested, with respect to the host language grammar CFG^H, using the *isComposable* and *conflictFree* analysis. If all extensions *independently* pass this analysis, then their composition is a CFG from which a conflict-free deterministic LR parse table can be generated. Of course, this analysis puts some restrictions on the type of syntax that can be added to a language as a composable language extension, but we have found these restrictions to be natural and not burdensome [14]. This analysis is used in Copper, the integrated parser and context-aware scanner packaged with Silver.

One of the primary contributions of this paper is a modular analysis for attribute grammar completeness and circularity-detection that is meant to provide the sort of assurances described in the second question above. The modular completeness analysis, called *modComplete*, provides the following guarantee:

$$(\forall i \in [1, n].modComplete(AG^H, AG_i^E)) \implies complete(AG^H \cup \{AG_1^E, \dots, AG_n^E\}).$$

2

This analysis has the same form as the modular determinism analysis described above in that it verifies the property of attribute grammar completeness independently on each of the attribute grammar specifications of language extensions. The guarantee is that the non-expert can direct Silver to compose host and extension specifications (that pass this analysis) knowing that *the resulting attribute grammar will be complete*. Thus an entire class of common attribute grammar errors can be solved by the extension developers, who have some understanding of language design and implementation, and this burden will not fall on the non-expert doing the composition of the extensions.

Additional contributions of the paper include the following.

- In specifying the modular completeness analysis we define a notion of *effective completeness* that is useful in higher-order attribute grammars [23] as well as with forwarding.
- Unlike the original (non-modular) completeness analysis for attribute grammars with forwarding [16, 1], our analysis distinguishes between missing equations and cycles in the attribute dependencies resulting in better error messages to the developer.
- We extend the completeness analysis to a modular circularity analysis.
- We evaluate the restrictions imposed by the modular analysis on the Silver-language source specification of our (bootstrapped) Silver compiler. This highly-modular specification was written before the analysis was developed. We find that the restrictions are not overbearing.

Paper contents: Section 2 provides needed background on attribute grammars, defines a simplified AG language over which the analyses are described, and defines the mechanism for computing flow-types of nonterminals in the AG. In Section 3 we present a modular analysis for effective completeness which we then extend in Section 4 to allow for more flexible organization of grammars and to include additional AG features found in full-featured attribute grammar specification languages such as Silver. Section 5 describes our experience in applying this analysis on the specification of Silver. In Section 6 we augment the analysis to also ensure non-circularity. Related (Section 7) and future work (Section 8) are discussed, followed by concluding remarks (Section 9).

2   Background

We begin with a broad overview of attribute grammar analysis, and then get more specific, with an example grammar and flow graphs later in the section.

Attribute grammars [11] are a formalism for describing computations over trees. Trees formed from an underlying context-free grammar are attributed with synthesized and inherited attributes, allowing information to flow, respectively, up and down the tree. Each production in the grammar specifies equations that define the synthesized attributes on its corresponding nodes in the tree, as well as the inherited attributes on the children of those nodes. These equations defining

the value of an attribute on a node may depend on the values of other attributes on itself and its children.

An attribute grammar is considered *complete* if there are no missing equations. That is, for all productions, there is an equation for every synthesized attribute that occurs on the nonterminal the production constructs, and for all children of the production, there is an equation for every inherited attribute that occurs on that child's nonterminal.

An attribute grammar is considered *non-circular* (and, if also complete, then *well-defined*) if on every possible tree allowed by the context-free grammar, there is no attribute whose value, as specified by the attribute equations, eventually depends upon itself. Knuth presents [11] an algorithm to ensure non-circularity in a complete attribute grammar. This algorithm is based upon constructing a graph for each production that describes how information flows around locally within that production. Alone, this is insufficient information: a production has no idea how its own inherited attributes might depend on its own synthesized attributes, nor can it know how its children's synthesized attributes might depend upon the inherited attributes it provides them. This global information is determined by a data flow analysis that results in a set for every nonterminal containing every possible flow of information between attributes on that nonterminal. Non-circularity can be checked using these sets.

Attribute grammars have been extended in a variety of ways. Higher-order attribute grammars [23] allow attributes to themselves contain trees that are as-yet undecorated by attributes. These trees are made useful by permitting productions to "locally anchor" a tree and decorate it, as if it had been a child. A child, however, is supplied when a tree is created, whereas these "virtual children" are defined by an equation during attribute evaluation that, of course, may have dependencies on the values of attributes, like any other equation. Higher-order attribute grammars amend the notion of *completeness* by requiring all inherited attributes to be supplied to these "virtual children," as well. The notion of *non-circularity* is also further extended, by requiring that the synthesized attributes of each "virtual child" have an implicit dependency on the equation defining that tree. These virtual children can potentially lead to the creation of an unbounded number of trees, and thus the nontermination of attribute evaluation.

Forwarding [16] was introduced to attribute grammars to allow for language extension. A language extension that introduces new productions combined with another that introduces new attributes on existing nonterminals presents a serious problem for completeness: the new attribute may not be defined on the new production. This is sometimes referred to as the *expression problem*.[1] If, however, those new productions *forward* requests for synthesized attributes without defining equations to *semantically equivalent* trees in the host language, the new attributes can simply be evaluated on that host tree instead and returned as the value of the attribute for the *forwarding* production, resolving the problem. Forwarding amends the notion of *completeness* by allowing a production that forwards to omit synthesized attribute equations, as they can instead be sup-

---

[1] `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`

$$
\begin{aligned}
D \; ::= \;\; & \texttt{nonterminal} \;\; n_{nt} \;\; \texttt{;} \quad\; | \;\; \texttt{synthesized attribute} \;\; n_s :: T \;\; \texttt{;} \\
| \;\; & \texttt{inherited attribute} \;\; n_i :: T \;\; \texttt{;} \quad | \;\; \texttt{attribute} \;\; n_a \;\; \texttt{occurs on} \;\; n_{nt} \;\; \texttt{;} \\
| \;\; & \texttt{production} \;\; n_p \quad n_{lhs} :: n_{nt} \; \texttt{::=} \;\; \overline{n_{rhs} :: T} \;\; \{\;\; \overline{S} \;\; \} \\
| \;\; & \texttt{aspect production} \;\; n_p \quad n_{lhs} :: n_{nt} \; \texttt{::=} \;\; \overline{n_{rhs} :: T} \;\; \{\;\; \overline{S} \;\; \} \\
S \; ::= \;\; & n_{lhs} \,.\, n_s \;\; \texttt{=} \;\; E \;\; \texttt{;} \quad | \;\; n_{rhs} \,.\, n_i \;\; \texttt{=} \;\; E \;\; \texttt{;} \\
| \;\; & \texttt{local} \;\; n_{local} :: T \;\; \texttt{=} \;\; E \;\; \texttt{;} \quad | \;\; n_{local} \,.\, n_i \;\; \texttt{=} \;\; E \;\; \texttt{;} \\
| \;\; & \texttt{forwards to} \;\; E \;\; \{\;\; \overline{A} \;\; \} \;\;\; \texttt{;} \\
A \; ::= \;\; & n_i \;\; \texttt{=} \;\; E \\
E \; ::= \;\; & n_{local} \quad | \;\; n_{lhs} \,.\, n_a \quad | \;\; n_{rhs} \,.\, n_a \quad | \;\; n_{local} \,.\, n_a \quad | \;\; \cdots \\
T \; ::= \;\; & n_{nt}
\end{aligned}
$$

Fig. 1. The language $\textsc{Ag}_f$

plied by the forward tree. Forwarding's necessary modifications to *non-circularity* roughly follow those of higher-order attribute grammars: the forward tree appears as a "virtual child" and all synthesized attributes on a forward tree have a dependency on the equation defining this tree. Forwarding introduces implicit "copy" rules for synthesized attribute equations the production is missing, as well as for any inherited attributes not supplied to the forward tree.

A problem for completeness identified by the forwarding paper, but also existing for higher-order attribute grammars, is the inconvenience of requiring *all* inherited attributes to be supplied. Frequently, only a subset of synthesized attributes are demanded, which in turn only require a subset of inherited attributes. This shows up frequently for forwarding, where a production may only use its own children to synthesize a "pretty print" attribute, relying on the forward tree for everything else (such as "translation.") This production would be required to supply its children with inherited attribute equations that are never used, merely to pretty print the tree. An amended notion of *effective completeness* of inherited attributes can be used instead: we require that all inherited attributes *needed to compute any accessed synthesized attribute* be supplied, instead of simply all of them outright. An *effectively complete* attribute grammar can compute non-circularity in the same manner as a complete one: if these equations are never demanded, they will never have an effect on flow graphs of a nonterminal, and can be ignored in the flow graphs of a production.

2.1 The language

The language defined in Fig. 1 describes a simplified attribute grammar language, based on our attribute grammar language, Silver, but with many orthogonal features (such as an indication of a starting symbol) omitted, and some introduced later in Section 4. It should generalize well to other attribute grammar languages that include forwarding. Declarations are represented by D. Attributes are declared separately from the occurs-on declarations that indicate what nonterminals (n_{nt}) the attribute (n_a) decorate. Semantic equations (S) can be supplied separately from declaring a production via *aspect productions*.

The possible equations include defining synthesized attributes (n_s) for the production's left hand side (n_{lhs}) and inherited attributes (n_i) for children (n_{rhs}) and locals (n_{local}). Local declarations allow for "locally anchoring" trees, as in higher-order attribute grammars. Finally, productions may forward, and provide equations to change the inherited attributes supplied to the forward tree, which are otherwise copied from the forwarding tree. Note that one restriction not reflected in the grammar is that a forward may not appear in an aspect production. Expressions (E) are largely elided from the language above. Only those expressions that induce dependencies in a production's flow graph are shown. As a result, even though referring to a child's tree (n_{rhs}) is a valid expression, it does not appear in E because a child tree is simply a value with no incurred flow dependencies. Similarly, any sort of function call expression induces no dependencies on its own, and simply aggregates dependencies from its component expressions.

We will write AG^H to indicate a host language, which should be a valid attribute grammar consisting of a set of declarations ($\overline{D}$.) We will write language extensions (also consisting of a set of declarations $\overline{D}$) as AG^E, and these grammars should be valid *in combination with* the host language they extend (i.e. $AG^H \cup AG^E$ is valid for each AG^E.) By validity, we mean certain properties about the grammars that we consider to be part of a "standard" environment and semantic analysis. For example, we will assume the grammars have all names bind properly and are type correct. We will assume that duplicate declarations of nonterminals, attributes, and productions are caught locally, and that if they occur in different grammars they are not duplicates but truly different symbols with different "fully-qualified" names based on the name of the grammar.

Fig. 2 shows an example of a small host language with two extensions. Note that one extension introduces a new production that forwards to its semantic equivalent in the host language (via De Morgan's laws), and that another extension introduce a new "translation" attribute for the productions in the host language. Both the `errors` and `java` attributes for the `or` production will be ultimately be computed by consulting the forwards tree.

The flow graphs for some of the productions of the composed grammars of Fig. 2 are shown in Fig. 3. Note that we use arrows to represent *dependencies* necessary to evaluate attributes, rather that using them to indicate the direction of information flow. A *flow type* [13] is a function $f_{nt} : syn \rightarrow \{inh\}$ that defines, for a nonterminal, what inherited attributes each synthesized attribute that occurs on that nonterminal may depend upon. A flow type can also be thought of as a graph, where edges are always from synthesized attribute nodes to inherited attribute nodes. A production's flow graph and the flow types for each nonterminal can be combined into a *stitched flow graph*. In the figure, the flow type for `Expr` is shown, along with the stitched flow graph for the production `not`. The flow types introduce edges between the attributes on the nonterminals of each child, virtual child, and forward in the stitched flow graph.

6

```
host grammar                                    or extension
nonterminal Expr;                               production or
synthesized attribute errors::[Msg];            e::Expr ::= l::Expr r::Expr
inherited attribute env::Env;                   { forwards to
attribute errors occurs on Expr;                    not(and(not(l), not(r))); }
attribute env occurs on Expr;
                                                java extension
production and                                  synthesized attribute java::String;
e::Expr ::= l::Expr r::Expr                     attribute java occurs on Expr;
{ e.errors = l.errors ++ r.errors;
}                                               aspect production and
production not                                  e::Expr ::= l::Expr r::Expr
e::Expr ::= s::Expr                             { e.java = l.java ++ "&&" ++ r.java; }
{ e.errors = s.errors;                          aspect production not
}                                               e::Expr ::= s::Expr
production var                                   { e.java = "!" ++ s.java; }
e::Expr ::= n::Name                             aspect production var
{ e.errors = lookup(e.env, n.lexeme);           e::Expr ::= n::Name
}                                               { e.java = n.lexeme; }
```

Fig. 2. An example of a host grammar for boolean propositions, with two extensions.

2.2 Flow type computation

Knuth's (corrected) algorithm for ensuring non-circularity can be thought of as computing a set of flow types for each nonterminal. For now, we are only concerned with computing a single flow type per nonterminal. The flow type computation is a function $flowTypes(\overline{D}) : nt \rightarrow syn \rightarrow \{inh\}$. Supplied with a set of declarations that form a valid attribute grammar, it results in a function mapping each nonterminal to a flow type. In principal, flow types could be computed using the standard non-circularity algorithm, by merging the set of flow types into just one for each nonterminal. It is significantly more efficient to compute them directly, with a slightly modified algorithm:

1. Local flow graphs for each production are computed.
2. Begin with an empty flow type for every nonterminal & synthesized attribute.
3. Iterate over every production. Produce a stitched flow graph for that production using the *current* set of flow types. If there are any paths from a synthesized attribute on the production's left hand side nonterminal to an inherited attribute on the same that are not yet present in the current flow type for that nonterminal, add them to the nonterminal's flow type.
4. Repeat until no new edges are introduced in a full pass over the productions.

We also have need to extend the domain of the flow type function ft_{nt} to track not just synthesized equation dependencies, but also those for forward equations. We will write $ft_{nt}(fwd)$ to refer to the dependencies potentially necessary to evaluate all forward equations for the nonterminal nt.
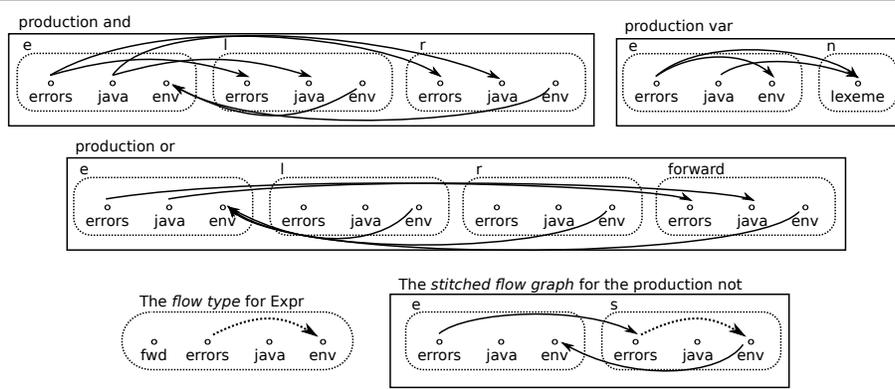
Fig. 3. Flow graphs for the grammar of Fig. 2. Also including the flow type of Expr, and an example stitched flow graph.

3 Modular flow analysis for completeness

The modular completeness analysis *modComplete* checks six properties of the host and each extension attribute grammar individually to ensure that the composition of the host and these extensions will be effectively complete. Two of these properties require the flow types for host and extension grammars to have been computed. The analysis *modComplete* is defined as follows:

$modComplete(AG^H, AG^E) \triangleq$
   $noOrphanOccursOn(AG^H, AG^E) \ \wedge \ noOrphanAttrEqs(AG^H, AG^E) \ \wedge$
   $noOrphanProds(AG^H, AG^E) \qquad \wedge \ synComplete(AG^H, AG^E) \ \wedge$
   $modularFlowTypes(flowTypes(AG^H), flowTypes(AG^H \cup AG^E)) \ \wedge$
   $inhComplete(AG^H, AG^E, flowTypes(AG^H \cup AG^E))$

Each of these checks is defined in turn below. In these discussions, we will use the notation "*n* is exported by AG_1 or AG_2" to mean that the symbol *n* is declared in the grammars AG_1 or AG_2. This ensures that when `export` statements are introduced in the extended analysis (Section 4) the language used below will still be a correct description of the requirements. We will also say that something is "in scope" if the information is available in the standard environment for a grammar, as described in Section 2.

No orphan occurs-on declarations: The check *noOrphanOccursOn* ensures that there will be no duplicate occurs-on declarations in the composition of the host and all extension grammars, denoted AG^{all}.

$noOrphanOccursOn(AG^H, AG^E)$ holds if and only if each occurs-on declaration "`attribute` *a* `occurs on` *nt*" in $AG^H \cup AG^E$ is exported by the grammar declaring *a* or the grammar declaring *nt*.

Every occurs-on declaration will have all potentially duplicate occurs-on declarations in its scope. This prevents, for example, an occurs-on relation being declared in an extension AG^E for an attribute and non-terminal that are both defined in the host AG^H. However, it still permits the occurs-on declaration if the nonterminal or the attribute are declared in the extension.

No orphan attribute equations: The check *noOrphanAttrEqs* ensures that there will be no more than one equation for the same attribute for the same production in AG^{all}.

$noOrphanAttrEqs(AG^H, AG^E)$ holds if and only if each equation $n.a = e$ in a production p is exported by the grammar declaring the (non-aspect) production p or the grammar declaring the occurs-on declaration "`attribute` a `occurs on` nt" (where n has type nt.)

Similar to the orphaned occurs-on declarations, this rule ensure that each equation must have all potential duplicate equations in scope. This relies on the orphaned occurs check: if two extensions can independently make the same attribute occur on the same nonterminal, in such a way that the standard environment cannot catch the duplicate occurs, then it also cannot catch the duplicate equations. Also note that this rule applies equally to synthesized and inherited attribute equations, and that we have not yet ensured there exists *at least* one equation, only ruled out the possibility of more than one.

No orphan production declarations: The check *noOrphanProds* ensures that extension productions forward, in order to allow forwarding to solve the problem its introduction is intended to solve.

$noOrphanProds(AG^H, AG^E)$ holds if and only if for each production declaration p in $AG^H \cup AG^E$ with left hand side nonterminal nt, the production p is either exported by the grammar declaring nt, or p forwards.

This rule is different from the previous two in that there's no choice of where a declaration can appear. The grammar declaring the nonterminal in effect declares a fixed set of productions that do not forward, and this set will be known to every extension grammar. As a result, a production is either in the host language AG^H, declared in the extension and forwards, or is declared in the extension and its left hand side is a nonterminal also declared in the extension.

Completeness of synthesized equations: The check *synComplete* ensures that for every production, an equation exists to compute every synthesized attribute that occurs on its left hand side non-terminal in AG^{all}.

> $synComplete(AG^H, AG^E)$ holds if and only if for each occurs-on declaration `attribute a occurs on` nt, and for each non-forwarding production p that constructs nt, there exists a synthesized equation defining the value of that attribute for that production.

This rule relies on the previous orphaned productions rule to ensure that all non-forwarding productions are in scope at the occurs declaration. It further relies on the previous orphaned equations rule (and thus, the orphaned occurs rule) to ensure that all potential equations for those non-forwarding productions are in scope, and thus we can check for their existence. Any production that forwards will be able to obtain a value for this attribute via its forward tree if it lacks an equation, and therefore they do not need checking.

The rules up to this point ensure synthesized completeness in a modular way. No set of composed extensions that satisfy the above rules could result in a missing or duplicate synthesized equation for any production in the resulting composed attribute grammar, AG^{all}. These rules critically rely on forwarding. Without forwarding, nonterminals are no more extensible than standard datatypes in ML or Haskell, in that new synthesized attributes are possible, but not new productions. We now turn to inherited attributes and flow types.

Modularity of flow types: The check *modularFlowTypes* ensures that all grammars will agree on the flow types. For occurrences in AG^H, an extension is not allowed to change the flow types. For those in AG^E, it ensures they depend upon those inherited attributes needed to evaluate forward equations, at a minimum.

> $modularFlowTypes(flowTypes(AG^H), flowTypes(AG^H \cup AG^E))$ holds if and only if given $ft_{nt}^H \in flowTypes(AG^H)$ and $ft_{nt}^{H \cup E} \in flowTypes(AG^H \cup AG^E)$,
>
> 1. For all synthesized attribute occurrences `attribute s occurs on` nt declared in AG^H, $ft_{nt}^{H \cup E}(s) \subseteq ft_{nt}^H(s)$
> 2. For all nonterminals nt declared in AG^H, $ft_{nt}^{H \cup E}(fwd) \subseteq ft_{nt}^H(fwd)$
> 3. For all synthesized attribute occurrences, s, declared in AG^E where nt is declared in AG^H, $ft_{nt}^{H \cup E}(s) \supseteq ft_{nt}^H(fwd)$.

The first two properties prevent an extension from modifying the flow types of host language occurrences, including the forward flow type. The purpose of the requirement on extension attributes is less obvious, but boils down to potentially needing to be able to evaluate forwards to get to the host language production on which this attribute is defined (in particular for productions from other extensions.) Our implementation deals with this last requirement by simply modifying the flow types, rather than raising an error if one runs afoul.

Effective completeness of inherited equations: The check *inhComplete* ensures that no evaluation of attributes will demand an inherited attribute that is missing
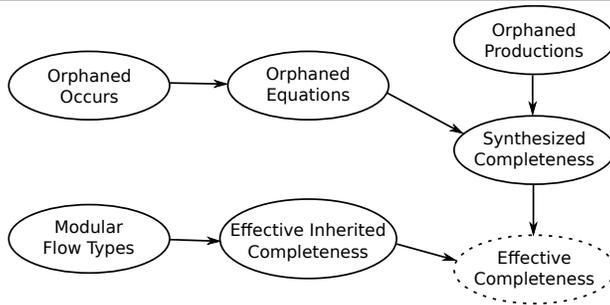
Fig. 4. The dependencies among the modularity analysis rules.

a defining equation. For each access of a synthesized attribute from a child or local, we ensure that a sufficient set of inherited attributes has been supplied to that child or local.

$inhComplete(AG^H, AG^E, flowTypes(AG^H \cup AG^E))$ holds if and only if for every production p in $AG^H \cup AG^E$ and for every access to a synthesized attribute $n.s$ in an expression within p (where n has type nt,) and for each inherited attribute $i \in ft_{nt}(s)$, there exists an equation $n.i = e$ for p.

This rule is sufficient to ensure that, when the host and extension are composed alone, no missing inherited attribute equations will be demanded. Together with the previous modularity rule for flow types, it's is also sufficient to ensure this property holds for AG^{all} because the flow type for host attribute occurrences cannot be changed by an extension. Finally, we can also be sure there are no duplicate inherited equations thanks to the earlier orphaned equations rule.

Fig. 4 summarizes the dependencies between the rules and how effective completeness is established. We achieve effective completeness for AG^{all} by ensuring modular effective inherited completeness and modular synthesized completeness hold for each extension individually, with no further checks necessary.

This analysis ultimately boils down to two major restrictions on extensions. First, the host language fixes a set of non-forwarding productions, and any extension to this language must ultimately be expressible in terms of these productions (via forwarding.) Second, for each host language synthesized attribute occurrence, the host language fixes a set of inherited dependencies. Our experience so far suggests that host languages are typically rich enough to express many interesting extensions, and many extensions that need inherited information can usually "piggy back" that information on already existing host language inherited attributes (e.g. the environment.) Further evaluation of the practicality of these rules will be presented in Section 5.

11

$$G \; ::= \cdot \;\; | \;\; \texttt{grammar} \;\; n_g \; \{ \; \overline{M} \;\; \overline{D} \; \} \; G$$
$$M \; ::= \texttt{import} \;\; n_g \;\; ; \;\;\; | \;\; \texttt{export} \;\; n_g \;\; ; \;\;\; | \;\; \texttt{option} \;\; n_g \;\; ;$$
$$| \;\; \texttt{export} \;\; n_g \;\; \texttt{with} \;\; n_{tg} \;\; ;$$

Fig. 5. Extending the language AG_f with a module system

4 Extending the analysis

We extend AG_f in two ways: first, by introducing a module system for it, and second, by introducing several other features found in Silver to the language that affect the modularity analysis.

4.1 A module system

So far we have referred to attribute grammars AG^H and AG^E as host and extension grammars, however this does not reflect the reality of developing large attribute grammars. In Fig. 5, we introduce a module system to AG_f. Each grammar is named, and consists of a set of module statements and a set of attribute grammar declarations. We will consider each module statement in turn.

The analysis presented in Section 3 is easily generalized to apply to acyclic import graphs between grammars. An import ($\texttt{import} \;\; n_g$) makes explicit the information available in the standard environment for a module, whereas we previously merely stated that extension grammars have in their environment their host language's symbols. To apply the modularity analysis, we consider every grammar to be an "extension" to the composition of all the grammars it imports, and we otherwise apply the analysis unchanged. The modularity rules are formulated such that being designated a "host" or "extension" confers no special status beyond what "is extending" and what "is being extended." A grammar that imports nothing satisfies all the modularity rules. In addition to allowing the host language to be broken up into multiple grammars, this allows extensions to make use of other extensions.

Exports ($\texttt{export} \;\; n_g$) allow grammars to be broken apart arbitrarily into different pieces, and be largely treated as a single grammar for the purpose of the analysis. For example, we might want to separate the concern of type checking from the host language, but type checking may not pass the analysis as an "extension." To allow this, we can have the host language grammar export the type checking grammar, essentially designating it part of the host. Again, the modularity rules do not require any changes, because we were already careful to word the rules to note when a symbol is "exported by" a grammar.

So far, however, this is still a limiting situation for host languages. Many languages have multiple potential configurations that cannot be reflected as extensions, for modularity reasons or simply because they conflict outright with alternative configurations. For example, GHC Haskell has many optional features that can be enabled, some of which cannot be activated together. To support these configurations, we introduce *options* to the language. An *option* ($\texttt{option}$

$$E ::= \texttt{ref}\ \ n_{local}\ \ |\ \underline{\texttt{ref}\ \ n_{lhs}}\ \ |\ \ \texttt{ref}\ \ n_{rhs}\ \ |\ \ E\ .\ n_a$$
$$|\ \ \texttt{case}\ E\ \texttt{of}\ \overline{p \to E_p}\ \ |\ \ n_v\ .\ n_a\ \ |\ \ \cdots$$
$$p ::= n_p(\overline{n_v})\ \ |\ \ \text{\_}$$

Fig. 6. Extending the language AG_f with references and pattern matching

n_g) declaration behaves identically to an export, when computing any of the requirements imposed by Section 3, but has no effect on the standard environment. This allows, for example, new non-forwarding productions to be introduced in a grammar that is not necessarily the host language, but still allows the modularity rules to ensure that any extensions account for their existence. This in turn necessitates another new feature, *conditional exports*, to allow an extension grammar to optionally include support for a feature that may or may not be in the host language (because it is an optional component.) A conditional export (`export` n_g `with` n_{tg}) is identical to a normal export of n_g, so long as the importing module also imports its triggering grammar (n_{tg}), such as an optional component of the host language. If not, the conditional export is ignored.

Finally, our last modification to the module system is to account for import cycles. The above generalization to arbitrary import graphs works until a cycle is encountered. If there is a cycle, then a "host" grammar will actually include the "extension" in its flow type computations, and thus won't flag that extension's violations. To handle cycles (and as a bonus, to compute flow types much more efficiently,) we compute flow types just once, globally. To ensure we still generate the same flow types, production graphs are partitioned into standard edges and *suspect edges*. Only standard edges are used when stitching production graphs, and computing updates to flow types. Suspect edges are generated from synthesized (and forward) equations that are not permitted to affect their corresponding flow type. (Note that this includes implicit synthesized equations generated by forwards.) During the flow type computation, the dependencies suspect edges would introduce are considered, and only those direct edges to LHS inherited attributes that are already in their corresponding flow type are admitted as a standard edge. In this way, these edges' valid dependencies have effect on the rest of the graph, while avoiding affecting their own flow type.

4.2 Additional language features

Silver also supports a version of reference/remote attributes [8, 3], and pattern matching [9]. References refer to trees that have already been given their inherited attributes elsewhere, whereas higher-order attributes refer to a tree that is "anchored" and supplied inherited attributes locally. References present a unique problem: the "decoration site" of the reference is unknown. It is not possible to know what inherited attributes where supplied, if any. We adopt an extremely conservative solution to this obstacle. We will refer to the set of *all* inherited attributes known to occur on a nonterminal nt **by the grammar that declares** nt as $ft_{nt}(ref)$. This particular choice isn't important, only that all grammars

13

will agree on a consistent set. Whenever a reference is taken (`ref` n_n, where n_n has type nt), we consider it to depend on $ft_{nt}(ref)$. Whenever an attribute is demanded from a reference (E . n_a where E is a reference to a nonterminal nt), we ensure that the flow type $ft_{nt}(a) \subseteq ft_{nt}(ref)$. Thus, the set of inherited attributes on a reference type is fixed by the host language. This could be extremely limiting, however in Section 5 we present some evidence that it is still quite workable.

Silver allows for pattern matching on trees, in a manner that respects forwarding [9]. In order to perform the pattern matching, therefore, we must be able to evaluate the forward equation for any production, and therefore the scrutinee must be supplied $ft_{nt}(fwd)$. As a bonus, the interaction of pattern matching with forwarding, combined with the orphaned production rules of Section 3, allows pattern matching expressions to ensure all possible cases are covered. Pattern matching is capable of extracting references to the children of a production, but in a manner that constitutes a known "decoration site," and thus we do not have to fall back on treating them like references (so long as reference is not otherwise taken.) Given a set of inherited attributes known to be supplied to the scrutinee i, for each case matching a production p, we can flow i through the production flow graph for p, and determine the set of inherited attributes that will be supplied to each child of p extracted as a pattern variable. For each pattern variable attribute access (n_v . a where n_v has type nt) we can ensure the flow type $ft_{nt}(a)$ is a covered by this set.

Silver has a number of other features that have relatively trivial effects on the modularity analysis: collection attributes, autocopy attributes, and newly-introduced default equations, see Section 5. The later two present no complications, and in fact are greatly simplified by the analysis: for grammars that pass the analysis, all implicit equations they introduce are statically known. Our notion of collection attributes do not allow for remote contributions, unlike those of Boyland [3]. Collections attributes are synthesized attributes that have two different kinds of defining equations: base and contribution. Base equations are treated identically to ordinary equations for the modularity analysis. Contributing equations are allowed to exist in places that violate the orphaned equations rule (as there are allowed to be any number of them), but are still subject to the inherited completeness rule and the host language's flow type.

Finally, Silver supports an alternative composition model that more resembles that of classes in object-oriented languages. *Closed* nonterminals, instead of having a fixed set of non-forwarding productions, have a fixed set of attributes instead, and non-forwarding productions may appear in any grammar. As an example, many languages will have a concrete syntax tree with a single synthesized attribute to construct an abstract syntax tree. This poses a potential serious annoyance for extensions, given the modularity rules: the extension might have to duplicate the forwards for the concrete and the abstract productions. Making concrete syntax nonterminals closed resolves the issue.

5 Evaluation of modular completeness analysis

To evaluate the analysis, and the practicality of writing specifications that satisfy the restrictions imposed, we have implemented the analysis and applied it to the Silver specifications for the Silver compiler. We chose to analyze Silver itself because it was one of the most complex attribute grammar specifications we have. The host language has an interesting type system, it includes several optional components that may or may not be included, it has several composable language extensions (some add new syntax, some add new attributes, some both), and it has a full translation to Java. It's also the Silver specification we use most, and as a result we believe it would have the fewest bugs.

The caveat for evaluating on Silver is that what is considered "host" vs "extension" is also under our control. To alleviate this concern somewhat, we briefly describe a few of the extensions, to demonstrate they are interesting and non-trivial. A "convenience" extension introduces new syntax that greatly simplifies making large numbers of similar occurs declarations, by allowing nonterminal declarations to be annotated with a list of them. A testing extension adds several constructs for writing and generating unit tests for the language specification. An "easy terminals" extension allows simple terminals to be referred to by their lexeme in production declarations (using 'to' instead of `To_kwd`, for example.) Finally, the entire translation to Java is implemented as a composable language extension.

A technical report documents all issues raised by the analysis and the changes made to address them [10]. A brief summary of those results is reported here.

Silver focuses specifically on language extension, and as a result, we had chosen not to implement the monolithic analysis, to better enable separate compilation. Without the modular analysis, we had simply gotten by without a static completeness analysis. One set of changes made in response to the analysis were expected: we found (and fixed) several bugs. The analysis found several legitimately missing synthesized and inherited attribute equations. It also found several productions that should have been forwarding, but were not.

Another positive set of changes improved the quality of the implementation, even if they did not directly fix bugs. We discovered several extraneous attribute occurrences that simply never had equations, and were never used either. Many uses of references were found to be completely unnecessary and eliminated. One particularly interesting change has to do with how concrete syntax specifications are handled in Silver. Silver's host language supplies a "standard" set of declarations for concrete syntax, while Copper-specific declarations are kept in a separate optional grammar. The analysis raised a simple error: the Java translation attributes for parser declarations were being supplied by the Copper grammar, which is a violation of the rules. The decision was made to move this parser declaration out of the host language and into the Copper optional grammar, and that it actually belonged there all along: it does, after all, generate a Copper parser.

Some parts of the analysis were motivated by our attempts to get Silver to conform to the restrictions. The specification of the Silver host language is broken

up into several grammar modules for standard software engineering reasons of modularity. Many of these modules are *not* meant to be considered composable language extensions. This motivated the introduction of the "option" module statement. We also found that we were abusing forwarding, using it as a way to define default values for attributes where the forwarded-to tree was not, in fact, semantically equivalent to the forwarding tree in the slightest. To resolve this, we introduced the notion of *default* attribute values as a separate concept, so these uses of forwarding could be eliminated.

There were two sorts of negative changes made to the Silver specification in order to make it pass the analysis. The first of these resulted from the conservative rules for handling reference attributes. Two sets of inherited attribute equations had to be supplied whose values are never actually used. In one case, a nonterminal representing concrete syntax information has two synthesized attributes, one for a normalization process and one for translation. These attributes have different inherited dependencies, but the analysis required the full set for both because the synthesized attributes internally used references.

The second sort of negative changes involved introducing workarounds for code that we already knew needed refactoring, but we did not want to fix, yet. In fact, in many of these cases, the analysis lead us to code that already had "TODO" comments complaining about a design for reasons unrelated to the analysis. The most significant of these is the use of a single nonterminal as a data structure to represent several different types of declarations in Silver (attributes, types, values, occurs-on, etc.) This is a legacy from when Silver did not have parametric polymorphism and needed to group all of these together into a single monomorphic type. To make the analysis pass, we introduced "error" equations for attributes that did not have sensible values otherwise (e.g. attributes for *value* declarations that do not apply to *type* declarations.) However, the use of "error" equations to make the analysis pass still provides a statically detectable indication that we are, in essence, making a temporary end-run around the analysis. These error equations are essentially a form of "technical debt" - legitimate problems that we will change later, but for various reasons decide not to do just yet. On the positive side, it lets the developer distinguish between bugs to fix now and changes to make later.

In our experience with Silver, the analysis found a few bugs, motivated us to fix a small number of design flaws, and introduced a relatively small amount of technical debt. The analysis also found problems that inspired some changes to Silver (*e.g.* options, defaults) to more easily write specifications that satisfied the restrictions. We found the use of "error" equations reasonable, as a way to document technical debt and refactorings that should be made at a later date (or in the worst case, an explanation of why the attribute really could never be demanded, despite the analysis indicating otherwise.) In the end, most of the changes necessary were to the host language itself, and the extensions then passed without further effort.

6   Extending the modular analysis to circularity analysis

So far we have focused only on ensuring *completeness* of the composed attribute grammar in a modular way. This involved making use of flow information that is typically used to ensure non-circularity, but we only computed a single flow type for each nonterminal instead of a set, as is normally done in circularity analyses. To ensure non-circularity, we will go back to calculating these flow sets once again.

In this section we define a modular non-circularity analysis. As in the modular completeness analysis, this analysis is performed independently on each extension to ensure non-circularity in the final composed grammar. This analysis, *modNonCircular*, is defined as follows:

$modNonCircular(AG^H, AG^E) \triangleq$
  $\quad modComplete(AG^H, AG^E) \wedge nonCircular(AG^H \cup AG^E) \wedge$
  $\quad modFlowSets(flowSets(AG^H), flowSets(AG^H \cup AG^E))$

The *modComplete* analysis is unchanged, and the analysis *nonCircular* is the standard non-circularity analysis for attribute grammars [11, 23, 16].

Modularity of flow sets: The check *modFlowSets* ensures that extensions do not introduce any flow types whose host language component is not already accounted for in the host language's flow sets. The extension can still introduce new flow types, so long as they differ only in edges from synthesized attribute occurrences declared in AG^E.

$modFlowSets(flowSets(AG^H), flowSets(AG^H \cup AG^E))$ holds if and only if for every $ft_{nt}^{H \cup E}$ in $flowSets(AG^H \cup AG^E)$, there is a ft_{nt}^H in $flowSets(AG^H)$ such that for every $s \in dom(ft_{nt}^H)$, $ft_{nt}^{H \cup E}(s) \subseteq ft_{nt}^H(s)$.

This rule is justified by a well-known optimization for computing flow sets, where flow types that are "covered" by another flow type in the same flow set can be discarded. All this rule does is ensure that all flow types generated by the extension (restricted back to the host language) can be discarded in this fashion, and consequently do not affect the non-circularity check.

This analysis, too, can handle the extensions introduced in Section 4, with the caveat that the conservative rules introduced for handling references may result in false positive circularities. Finally, while we have not yet evaluated this analysis in practice, we believe the major potential problem for it is ensuring that the extension developer can understand the resulting requirements. The graphs that are contained in the flow sets are not necessarily subject to easy intuition the way flow types are.

7   Related Work

Knuth introduced attribute grammars [11] and provided a circularity analysis. In presenting higher-order attributes, Vogt *et al.* [23] extended Knuth's completeness and circularity analyses to that setting. Reference and remote attributes do

not have a precise circularity analysis [3], as the problem is undecidable. Completeness in these settings is simply a matter of using occur-on relationships to check for the existence of equations for all of the required attributes. With forwarding, flow-analysis is used to check completeness and thus a *definedness analysis* that combines the check of completeness and circularity was defined [16, 1]. This analysis used *dependency functions* instead of flow graphs in order to distinguish between synthesized attributes that depend on no inherited attributes and those that cannot be computed because of a missing equation or circularity, and thus conflate this two types of errors. All of these are non-modular analyses.

Saraiva and Swierstra [13] present *generic attribute grammars* in which an AG has grammar symbols marked as *generic* and not defined in the grammar. Composition in this model is the instantiation of these generic symbols with specific ones. Here, flow-types can be computed on the generic grammar being imported and then used when flow analysis is done on the instantiating/importing grammar. This composition model, however, is very different from the language extension model described in Section 1. It does not allow for multiple independent extensions to be composed, except by first merging them into a single extension, on which the analysis must then be performed, effectively making it monolithic.

In AspectAG, Viera *et al.* [20] have shown the completeness analysis can be encoded in the type system of Haskell. However, this analysis is again performed at the time of composition (by the type checker) and is thus a monolithic analysis.

Current AG systems such as JastAdd [6] and Kiama [15] do not do static flow analysis but, like previous versions of Silver, instead provide error messages at attribute evaluation time that indicates the missing equation or circularity. An extension writer can write test cases to test his or her specification and perhaps find any lurking problems, but this does not provide any assurances if independently developed grammars are later composed.

8   Current and Future Work

We would like to apply these analyses to other attribute grammars specified using Silver to further evaluate their usefulness. One is possibility is ABLEJ our extensible specification of Java with many different language extensions [18].

We plan to evaluate the practicality of the circularity analysis. We have not done this on the Silver compiler because Silver is a lazily evaluated language, and our compiler implementation has known "circularities" between attributes that are, in fact, well-defined thanks to laziness. To apply the non-circularity analysis to Silver, we will have to determine whether these can be eliminated or the analysis can be extended to somehow deal with these apparent circularities.

The completeness and circularity detection analyses do not check that an unbounded number of attributable trees will not be created. This is the third component of *well-definedness* identified by Vogt *et al.*in their original work on higher order attribute grammars [23], but dropped as a well-definedness require-

ment in his Ph.D. dissertation [22]. Other members of our group have separately explored a termination analysis based on term rewriting systems [12].

9 Conclusion

We have presented a *modular* analysis for completeness and non-circularity in attribute grammars. This differs from prior analyses in that languages extensions are checked prior to being composed together, independently of each other, with no checks necessary after composition to ensure these properties. In the extensible language scenario described here we do not have the luxury of a monolithic (composition time) analysis since the person composing the extension grammars is not expected, nor required, to understand attribute grammars, or even know what they are. For extensible languages and extensible language frameworks to be useful to most programmers we believe that these sort of *modular* analyses are critical in order to ensure that the composition of independently developed language extensions "just works."

The analyses do not, and cannot, check that the *forwards* tree on a production is semantically equivalent to the tree doing the forwarding. The language developer is stating, by using the forwarding mechanism that the two are semantically equivalent. If they are not then the attribute values returned from the forwarded-to tree may not be correct. However, a misuse of forwarding in this fashion is a problem with one specific extension, rather than a problem arising from the composition of extensions (though it may only be exposed by a composition of extensions.)

The two questions raised in Section 1 are related to ease of composition of grammars by the non-expert programmer, and not about ease of specification of the language or language extensions by the language developer. The restrictions imposed by the modular analysis are designed to ease the work of the non-expert programmer. That said, we certainly do want to provide as much support as possible to the language developer. The extensions to the analysis to cover all the language features of Silver and to provide new features such as options and attribute defaults are there to make conforming to the restrictions easier. In our experience with Silver, these imposition of the restrictions is more than offset by the strong guarantees that the analyses provide.

References

1. Backhouse, K.: A functional semantics of attribute grammars. In: Proc. of TACAS. LNCS, vol. 2280, pp. 142–157. Springer (2002)
2. Bird, R.S.: Using circular programs to eliminate multiple traversals of data. Acta Informatica 21, 239–250 (January 1984)
3. Boyland, J.T.: Remote attribute grammars. J. ACM 52(4), 627–687 (2005)
4. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: Proc. of OOPSLA. pp. 365–383. ACM Press (2004)

5. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: Proc. of OOP-SLA. pp. 1–18. ACM (2007)
6. Ekman, T., Hedin, G.: The JastAdd system - modular extensible compiler construction. Science of Computer Programming 69, 14–26 (December 2007)
7. Erdweg, S., Rendel, T., K astner, C., Ostermann, K.: SugarJ: Library-based syntactic language extensibility. In: Proc. of OOPLSA. ACM (2011)
8. Hedin, G.: Reference attribute grammars. Informatica 24(3), 301–317 (2000)
9. Kaminski, T., Van Wyk, E.: Integrating attribute grammar and functional programming language features. In: Proc. of 4th the Intl. Conf. on Software Language Engineering (SLE 2011). LNCS, vol. 6940, pp. 263–282. Springer (2011)
10. Kaminski, T.: Evaluation of modular completeness analysis on Silver. Tech. Rep. 12-024, University of Minnesota, Department of Computer Science and Engineering. (2012), available at `http://melt.cs.umn.edu/pubs/kaminski12tr`
11. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2(2), 127–145 (1968), corrections in **5**(1971) pp. 95–96
12. Krishnan, L., Van Wyk, E.: Termination analysis for higher-order attribute grammars. In: Proc. of 5th the Intl. Conf. on Software Language Engineering (SLE 2012). LNCS, vol. 7745, pp. 44–63. Springer (2012)
13. Saraiva, J., Swierstra, D.: Generic Attribute Grammars. In: 2nd Workshop on Attribute Grammars and their Applications. pp. 185–204 (1999)
14. Schwerdfeger, A., Van Wyk, E.: Verifiable composition of deterministic grammars. In: Proc. of PLDI. ACM (June 2009)
15. Sloane, A.M.: Lightweight language processing in Kiama. In: Proc. of the 3rd Summer School on Generative and Transformational Techniques in Software Engineering III (GTTSE 2009). LNCS, vol. 6491, pp. 408–425. Springer (2011)
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. Van Wyk, E., Krishnan, L., Schwerdfeger, A., Bodin, D.: Attribute grammar-based language extensions for Java. In: Proc. of ECCOP. LNCS, vol. 4609. (2007)
19. Van Wyk, E., Schwerdfeger, A.: Context-aware scanning for parsing extensible languages. In: Intl. Conf. on Generative Programming and Component Engineering, (GPCE). ACM Press (October 2007)
20. 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)
21. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. In: Rewriting Techniques and Applications (RTA'01). LNCS, vol. 2051, pp. 357–361. Springer (2001)
22. Vogt, H.: Higher order attribute grammars. Ph.D. thesis, Department of Computer Science, Utrecht University, The Netherlands (1989)
23. 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 (1989)