Composable Semantics Using Higher-Order Attribute Grammars

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Lijesh Krishnan Manjacheri

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Eric Van Wyk

November, 2012

Acknowledgements

First and foremost, I would like to express my heartfelt gratitude to my adviser Eric Van Wyk for his guidance, his superhuman patience and his pushing me at every stage to work harder and better. I would like to thank Professors Mats Heimdahl, Gopalan Nadathur and Wayne Richter for agreeing to serve on my committee. I am grateful to Professor Nadathur for his helpful comments and suggestions on the initial drafts of this dissertation, which have resulted in significant improvements. I thank Derek Bodin, Jimin Gao, Eric Johnson, Paul Johnson, Ted Kaminski, Yogesh Mali, August Schwerdfeger and other members of the MELT group for their assistance and the many useful discussions over the years. Thanks to Betsy, Katie, Kitti, Prashant and Yohaan for friendships that sustained me over this long and winding road.

And Colin, Mom, Sis, I love you all.

Dedication

To my Dad.

ABSTRACT

Ideally, programmers could make use of domain-specific knowledge and program using constructs that implement abstractions in their problem domains, and obtain domain-appropriate feedback. Further, new functionality could be added to existing languages incrementally, so that programmers could choose features appropriate to their problem domains, retrieve their specifications, and compose them automatically with an existing "host" language to generate a compiler for an extended version of the language. The library model of language extensibility separates out the stages of host language specification, extension development, and extension composition, to make the process of language extension modular. The model is targeted at writing composable semantic specifications, and emphasizes the importance of static analyses performed at extension development time to flag potential conflicts between extensions during composition.

Silver is a general-purpose, high-level framework that implements the library model of extension development. The declarative Silver specification language can be used to specify the host language concrete syntax and semantics, and independently, the extension syntax and semantics, the latter often in terms of the host language's semantics. The problem of brittle concrete syntax specifications is handled by the associated Copper tool that uses context-aware scanning to generate a parser for the extended language. The problem of specifying host language and extension semantics (such as error checking and source-level transformations) is handled via the evaluation on syntax tree nodes of functions written in the higher-order attribute grammar formalism. There are challenges to designing language semantics (such as types, environment and error-reporting) for a non-trivial host language to allow extension writers to write useful extensions that compose with other independently written extensions. An issue when generating sound and terminating generated compilers in a higher-order attribute grammar framework such as Silver is the potential for improper termination of attribute evaluation during the execution of the generated compiler.

This dissertation makes two contributions toward writing composable specifications for programming language semantics. It first looks at how a declarative, attribute grammar-based tool (extended with features such as forwarding, aspect productions and collection attributes) can be used to write host language specifications whose type systems and environments can be conveniently accessed and modified by extensions. To this end, it describes ableJ, an extensible host language specification for Java 1.4 that generates front-end translators from extended code to valid Java 1.4 code. Second, it presents a static analysis on higher-order attribute grammars that detects non-terminating tree creation during attribute evaluation. Combined with the higher-order circularity test, this analysis provides extension writers and programmers with a guarantee that the generated compiler will not fail to terminate on account of improper attribute evaluation. We have run the analysis on the ableJ host language and its extension specifications to demonstrate that the analysis is powerful enough to show termination of non-trivial grammars.

Contents

List of Tables

List of Figures

Chapter 1

Composable Extensions Using Higher-Order Attribute Grammars

1.1 Introduction

In this chapter, we first describe the overarching goal of this research: composable language extensions. We look at the challenges involved and at existing approaches. We then give a description of the library model of language extensibility and the Silver extensible language tool. We end by summarizing the specific contributions of this dissertation in the area of composable semantic specifications using higher-order attribute grammars.

1.1.1 Composable Language Extensions

When faced with a programming problem in a particular domain, programmers must overcome the semantic distance between high-level domain-specific abstractions, and the non-domain-specific general constructs available in most programming languages. Ideally, programmers could make use of domain-specific knowledge, and program using constructs that implement abstractions in the problem domain, obtaining domain-appropriate feedback. This would result in code that was easier to write and maintain. The process of adding functionality to existing languages could preferably be done incrementally. Programmers would be able to choose the programming features appropriate to their problems, retrieve specifications that implement these features, and compose them automatically with an existing language, thereby generating a compiler for the desired extended version of the language. This is similar to how multiple libraries can be imported and safely used to add functionality in standard programming languages. For example, a programmer who wishes to solve a problem involving both complex numbers and relational databases, would import a complex type extension and an SQL extension into Java and thereby create a customized extended version of Java with features appropriate to both his problem domains.

1.1.1.1 The Expression Problem

The expression problem is the problem of adding new functionality as well as new constructs to an existing language, without recompiling existing code [1]. Most extensible language tools handle one aspect of this problem. For example, macros allow for new constructs to be added. Existing semantics can be defined for these new constructs via translations to equivalent code in the base language. Object-oriented languages are particularly suited to the problem of adding new constructs. Inheritance can be used to define new forms of existing classes, and existing functionality for these constructs can be defined either explicitly or in terms of existing classes (though not both). Functional languages, on the other hand, are suited to the problem of adding new functionality (such as new forms of error-checking or a new translation).

In the context of language extensions, the expression problem has two facets:

- modularity: the basic problem of how new functionality and new constructs may be added, and
- composability: the problem of writing extensions that, though written independently, compose.

Multiple class definitions can be combined into a new hierarchy, as when Java libraries are imported and used by programmers. Similarly, multiple function definitions can be combined easily. But the problem examined here is that of composability in contexts where both new constructs and new functionality can be added. Questions arise as to how new functionality would be defined for any new constructs when multiple extensions are composed. Existing approaches to extensibility handle the issue of composable extensions in different ways.

1.1.1.2 Using General-Purpose Programming Languages

As mentioned above, general-purpose programming languages do provide facilities for developing new abstractions. These include higher-order functions, classes, generics and parametric polymorphism. These may implement the functionality of a desired abstraction but do not provide new language constructs with new and more domain-appropriate syntax. Neither do they allow for specifying static domain-specific semantic analyses. However, such languages are widely available, well-supported and have efficient compilers. They are therefore appropriate for one-off solutions.

1.1.1.3 Domain-Specific Languages

Domain-specific languages (DSL) do provide syntax for new domain-specific constructs, in addition to new functionality. These new constructs raise the level of abstraction to that of the domain and thereby reduce the semantic gap between general programming languages and the specific needs of the programmer. Further, unlike general-purpose languages, DSLs can perform domain-specific optimizations and other static semantic analyses. Examples of DSLs [2] include the Structured Query Language (SQL) for retrieving and manipulating data from relational databases [3], the hardware description language Verilog for modeling and analysing electronic systems [4], and MATLAB for mathematical computations such as operations on matrices [5]. The drawback of the DSL approach is that it is often not economically viable to construct entirely new languages and compilers with associated tools for a given domain if the programmer base is not large [6]. And even if DSLs are available, it is often the case that a given programming problem spans more than one domain, which compounds the viability issue.

1.1.1.4 Other Approaches to Language Extensibility

Early approaches to language extensibility such as embedded domain-specific languages and traditional macros [7, 8] provided limited facilities for semantic analyses on new constructs, and little domain-specific feedback to the programmer. Further, they were restricted in their scope because they either added language features from specific domains, or only handled certain aspects of language extensibility such as adding new syntax or source-level optimizing transformations. Other, more recent approaches to language extensibility such as JavaBorg [9] and JastAdd [10] are more powerful but focus on modular syntactic and semantic specifications. They do not deal with the problem of automatic composition of extension specifications. A more extensive survey of related work in the area of language extensibility is provided in Section 6.1.

1.1.2 The Library Model of Language Extensibility

Ideally, extending languages with new constructs would be as easy as importing libraries. A programmer with no implementation knowledge could import independently written specifications into a host language compiler, and expect that in most cases these extensions would work together. Further, from the programmer's point of view, extensions would be like plug-ins: easy to use, safe and with features that have the same look-and-feel as constructs in the host language.

The library model of language extensibility separates out the stages of host language development, extension development, and automatic extension composition, to make the process of language extension modular. It separates out the roles of host language specification writer, extension writer and programmer. Such a model allows for better communication and division of labor among these distinct actors, to solve a problem that traditional compiler development does not address. These roles are associated with varying degrees of compiler expertise, domain-specific knowledge and programming needs. This distinction is similar to the difference between library writers, and the programmers who import and use their libraries. The host language and extension writers must possess a fair amount of programming language design and implementation knowledge. But if extensions can be safely and automatically composed, then the programmer could select the list of extensions he desires, and then automatically generate the compiler for the extended version of the host language. He would be spared by the modular nature of the model and the composability of the extensions from having to deal with low-level details of the interactions between the new constructs and the host language compiler.

Figure 1.1 shows the various components of the library model of language extensibility. It represents a situation where an extensible language tool is used by a host

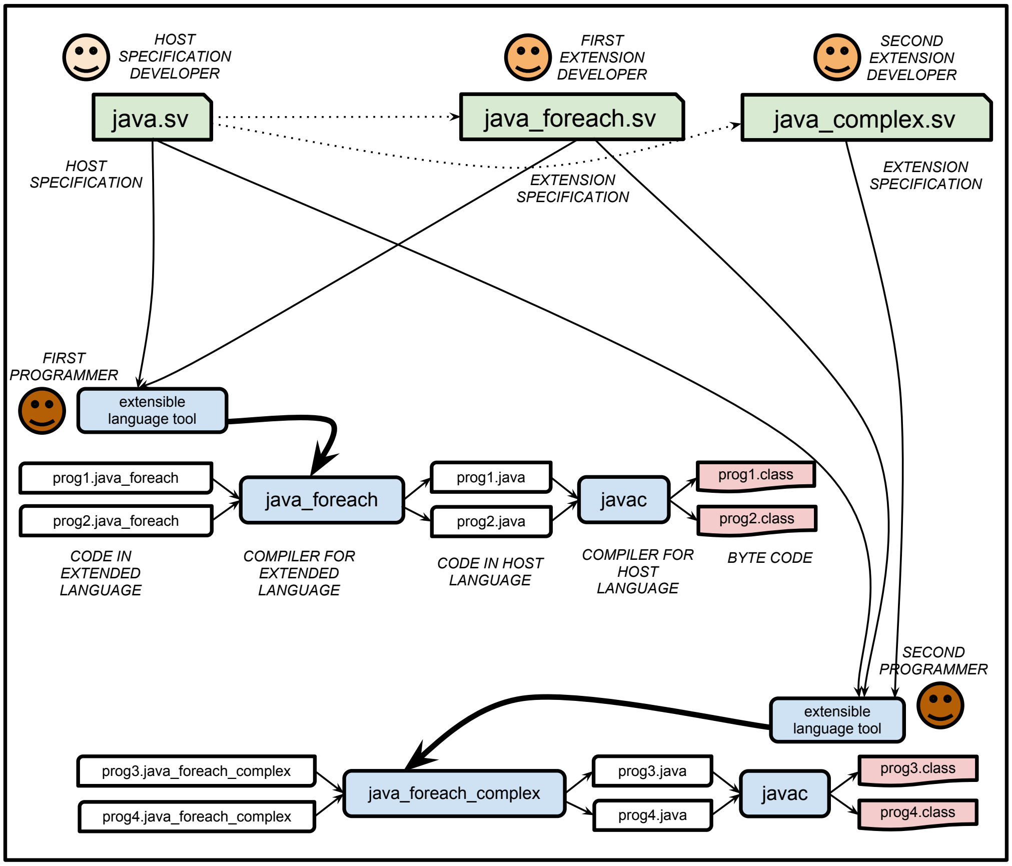


Figure 1.1: An overview of the actors, specifications and code in the library model of language extensibility.

```
complex[] coll;
...
for (complex c:coll) {
    System.out.println (c + 2.7);
}
```

```
Complex[] coll;
...
for (int i = 0; i < coll.length; i++) {
    Complex c = coll[i];
    System.out.println (new Complex (c.real () + 2.7, c.imag ()));
}
```

Figure 1.2: Code written in an extended version of Java 1.4, and equivalent pure Java 1.4 code.

specification developer, two extension developers, and two programmers. The figure shows the different actors, specifications and code written and generated at various points within the framework, when a pre-existing extensible specification of Java 1.4 is extended independently by two extension developers. The first feature adds the Java 1.5 enhanced `for` loop that provides syntax for iteration over arrays and collections. The second adds complex numbers as a primitive type and overloads operations such as addition to allow the programmer to use them directly on complex values. Figure 1.2 shows an example of code written in a version of Java 1.4 extended with these features, along with equivalent code in pure Java 1.4.

We now look at the tasks and challenges involved in these three stages.

1.1.2.1 The Host Language Specification

In the library model of language extensibility, the syntax and semantics of the host language are defined in a host language specification. In Figure 1.1, a host language specification developer has written a specification named `java.sv` for the Java 1.4 language. In this figure, each self-contained specification is represented as a single file with the extension `.sv`. Actual specifications usually consist of multiple files.

The host language specification writer makes important choices while designing the host language specification based on how it is likely to be extended by potential extension writers. He must think about what aspects of the host language and the compilation process would likely be accessed by extension writers. Aspects such as the host language's concrete syntax, type system and environment should be implemented to allow for new constructs and new analyses to be easily added. Access should be provided through an interface that allows for modular extensions, but also provides some measure of safety so that extensions are likely to compose.

1.1.2.2 Extension Specifications

Extension developers with domain expertise write extension specifications that define domain-specific language features with new concrete syntax and semantic analyses. Extension syntax and semantics are specified declaratively via self-contained extension specifications. In Figure 1.1, two separate extension developers have written extension specifications to the Java 1.4 host language. The first developer's specification `java_foreach.sv`, adds a Java 5-style enhanced `for` loop. The second specification, `java_complex.sv`, adds complex numbers as a primitive type to Java 1.4. Both these extensions have definitions that depend on the host language specification. This is indicated in the figure by dotted arrows.

To aid in modularity, extensions that add new constructs are expected to specify how equivalent host language code such as that shown in Figure 1.2 is to be generated. Thus in the library model, generated compilers are effectively source-level translators from extended to host language code. The generated host language code can be compiled with traditional compilers to generate valid executables. Ideally, extensions would provide efficient translations to valid host language code, as well as useful feedback to the programmer in the form of high-level domain-specific error messages. For example, if an attempt is made to use the enhanced `for` loop on a non-iterable value, a compile-time error would be reported indicating this fact. This would be more useful than a message reporting an error on generated code that used the Java 1.4 `for` construct. Obtaining extension-level feedback would shield programmers from having to examine and debug generated code.

1.1.2.3 Composing Host and Extension Specifications

The programmer would compose the host language specification and a set of extension specifications to automatically generate a compiler for the host language extended with the features defined by the composed extensions. Programmers with needs spanning multiple domains could custom build their programming languages by retrieving the appropriate extensions and composing them with the host language specifications. In Figure 1.1, two programmers with different programming needs specify two different extended versions of Java 1.4 by specifying which extension specifications are to be composed with the host language by the extensible language tool to generate the required translators. The programmers will use these generated compilers to translate the source code they write to pure and valid Java 1.4 code.

The first programmer creates a compiler for Java 1.4 with the enhanced `for` loop by composing the `java_foreach.sv` extension specification with the `java.sv` host language specification (indicated by solid arrows). The extensible language tool generates the executable `java_foreach` which translates the extended Java 1.4 programs `prog1.java_foreach` and `prog2.java_foreach` to the equivalent pure Java 1.4 programs `prog1.java` and `prog2.java`. The second programmer desires a version of Java 1.4 extended with both the enhanced `for` and complex number extensions. He composes both extension specifications with the host language specification. The generated executable `java_foreach_complex` translates the programs `prog3.java_foreach_complex` and `prog4.java_foreach_complex`, written in the extended language, to the equivalent pure Java 1.4 programs `prog3.java` and `prog4.java`. These are similar to the code fragments shown in Figure 1.2. The standard `javac` compiler can then be used to compile the generated Java programs into executable byte-code.

1.1.3 Silver: An Attribute Grammar Based Language Extension Tool

Silver [11] is a general-purpose, high-level framework that implements the library model of extension development. The declarative Silver specification language can be used to specify the host language concrete syntax and semantics, and independently, the extension syntax and semantics, the latter often in terms of the host language's semantics. Given a host language specification and a set of extension specifications, the Silver tool constructs a compiler for the desired extended version of the host language. The problem of brittle concrete syntax specifications is handled by the associated Copper tool that uses context-aware scanning to generate a parser for the extended language. The problem of specifying host language and extension semantics (such as error checking and source-level transformations) is handled via the evaluation, on the nodes of the constructed syntax tree, by Silver's attribute grammar evaluator-generator, of functions written in the higher-order attribute grammar formalism. A more detailed description of Silver is provided in Chapter 3.

1.1.3.1 Extensible Concrete Syntax via Copper

The availability of more intuitive syntax is a major incentive for the adoption of new domain-specific programming constructs. But there are challenges to developing extensible syntax specifications that can later be augmented with new constructs. There are issues with generating the scanner and parser for an extended version of a language, given a high-level declarative specification of host language and extension syntax. Parse tables generated by traditional LR-style parser generators are brittle in terms of whether shift and reduce conflicts emerge on the addition of new concrete productions [12]. There are also issues when composing lexical syntax specifications. A scanner cannot be generated if two extensions assign the same lexeme to different tokens. This is true even if syntactic context from the parser can be used to perform disambiguation, so that different lexemes can be assigned to the same input strings. It would be beneficial, therefore, for an extensible language framework to allow for such information to be passed from the parser to the scanner.

In Silver, the problem of composing concrete and lexical syntax is solved by the associated Copper tool, an integrated parser and context-aware scanner generator developed by August Schwerdfeger [13, 14, 15], a member of the author's research group. Copper reads in Silver declarations for grammar terminals, lexical classes, productions, precedences and associativities. It generates an integrated parser-scanner that performs scanning and parsing in tandem on source code in the extended version of the host language. Copper uses parser context during lexical disambiguation, and can therefore construct parsers for a larger class of languages than traditional parser generators. It employs a modified LR-style algorithm that uses information about the set of valid symbols at a given parser state to perform lexical disambiguation. Copper also includes a modular analysis to verify that the concrete syntax definitions in an extension specification follow certain (reasonable) restrictions which guarantee successful composition with other well-behaved extensions with respect to parser generation.

This discussion of the concrete syntax aspects of Silver is included to give a complete picture of the Silver framework. The author notes that this is not his work, but that of another member of the MELT group.

1.1.3.2 Modular Semantics via Higher-Order AGs

The problem of specifying and composing host language and extension semantics, is handled in Silver by its attribute grammar system. Attribute grammars [16] (described in Chapter 2) specify language semantics by associating values known as attributes to the nodes of program syntax trees. These hold semantic values such as types and errors. Attribute grammars extend context-free grammars with a set of functions, known as attribute definitions, on each production. Once the parser constructs a valid syntax tree, each node's attribute instances are computed by retrieving and evaluating attribute definitions from the appropriate productions. In higher-order attribute grammar frameworks [17] such as Silver, the set of possible attribute values is expanded from primitive values to include higher-order tree values that are valid syntax sub-trees themselves. This is useful in specifying analyses such as type-checking and code generation.

Thus, in Silver, host language semantics is specified as a higher-order attribute grammar, which is evaluated using Silver's attribute grammar system. Host language and extension semantics such as pretty-printing and type and binding checking are specified via attribute grammar fragments written using Silver's attribute grammar specification language.

The Advantages of Specifying Semantics Using HOAGs There are many different approaches to specifying language semantics. Each of these has its own set of challenges with regard to extensibility. Object-oriented frameworks, for instance, are well-suited to situations in which new constructs are added to an existing host language. Functional frameworks, on the other hand, are more appropriate in cases where new analyses are specified for existing constructs, but where no new constructs are added. Ideally the approach we employ would be well-suited to expressing programming language semantics via self-contained modular specifications that were easily composable. Modularity and composability are more likely if the host language semantics is specified using a high-level declarative framework.

The syntax-directed nature of attribute evaluation makes it a useful formalism for analysing code and performing tasks such as translation, code optimization and error-checking, and also for adding new features and analyses. Self-contained extension specifications can be written as separate grammar fragments, which can be statically analysed before being composed with the host language and other extensions to generate an evaluator for the extended language. Further, there are existing attribute grammar analyses and tools for constructing efficient evaluators. Using a well-defined, declarative and high-level formalism such as attribute grammars aids in the writing of modular and composable host language and extension semantic specifications. One drawback of attribute grammar frameworks is that attributes are not a convenient means to specify non-syntax directed analyses such as data-flow analyses on program control-flow graphs. Further, there is no guarantee that the attribute evaluation process will always terminate normally, an issue explored below.

1.1.3.3 Composing Host Language and Extension Modules

The Silver declarations for a host language's or an extension's lexical and concrete syntax, and its abstract syntax and semantics, are grouped into Silver files which are further organized into grammar modules. Modules group together related declarations such as the attribute grammar fragments that define the semantics of a host language, or the terminals, non-terminals and context-free grammar productions that define the concrete syntax of the host language or of a new construct added by an extension. Given a valid combination of specifications, the Silver tool composes the host language and extension concrete syntax and semantics, and generates an executable compiler for the desired extended version of the language. The generated compiler parses programs in the extended language, performs semantic analyses, and translates code in the extended language to valid host language code.

1.1.3.4 Static Extension Analyses to Ensure Composability

While modularity in extension specification is useful, composability is even more desirable. But independently developed extensions may not be compatible with respect to syntax or semantics. For example, two extensions may independently compose validly with the host language, but add new concrete syntax productions which when combined, result in LR-parse table conflicts. Similarly, two extensions may add sub-typing relations that independently work with the host language, but when composed together, introduce circularities in the host language type hierarchy. Thus it is essential for an extensible language framework to provide analyses on specifications to prevent problems during composition. A statically validated extension specification could be freely composed with its host language's specification and with other validated extension specifications.

Static specification analyses in Silver are performed at various stages of the framework and range from checks on the concrete syntax, to implementation-specific attribute grammar analyses. Some analyses detect problems modularly at extension development time, guaranteeing extension writers that their specifications will compose properly with other validated extension specifications. Such modular analyses detect problems at a stage when they can be solved by the extension writer, who presumably has the required domain knowledge and compiler expertise. For example, as mentioned in Section 1.1.3.1, Copper verifies that the concrete syntax definitions in an extension specification follow certain restrictions which guarantee successful composition (with respect to parser generation) with other well-behaved extensions [14]. Another example of a modular analysis is Silver's well-definedness analysis for attribute grammars [18].

Less useful than modular analyses are monolithic analyses performed at composition time on the extended compiler specification. These analyses flag incompatible extensions, but at a point where programmers would likely not have the access or expertise needed to resolve any conflicts, say by modifying the host language or extension specification. An example of such an analysis is the monolithic higher-order attribution termination analysis presented in Chapter 5 (which also describes a modular version of the analysis).

As extension complexity increases, the likelihood that static analyses will detect most run-time problems decreases. The extension writer bears a larger share of the responsibility of ensuring safety and correctness. While we have analyses that provide certain useful guarantees on the generated compiler (for example with respect to concrete syntax composition or attribute grammar evaluation termination), it can be difficult to write composable extension specifications for languages like Java. The host language must therefore be designed with modular composability in mind.

1.1.4 Challenges to Writing Modular Composable Semantics

1.1.4.1 Writing Composable Semantic Specifications

The Silver tool provides a foundation for handling the expression problem. Higher-order attribute grammars extended with features such as forwarding, aspect productions and collection attributes are an appropriate tool for adding new constructs (via new productions) and new functionality to existing languages (via new attributes and aspect productions). With simple macro-type extensions, such as adding a Java 1.5 style enhanced `for` loop to Java 1.4, this is sufficient to build composable extensions. Simple rewriting to equivalent host translations can be used to implement them. But as extension complexity increases, it is not clear how extensions can access and modify the host language's semantic analyses in a way that does not clash with other extensions doing the same. There are challenges to designing basic aspects of programming language semantics (such as types, environment and error-reporting) for a non-trivial host language in a way that allows extension writers to write useful extensions that compose with other independently written extensions. Extensions may specify domain-specific analyses for the constructs they add or define new analyses for existing features. Further, it would be helpful for extensions to be able to re-use host language semantics, as this would increase modularity and composability.

Extensions must be able to add new statements, expressions and types with the same look-and-feel as constructs in the host language, so that they fit comfortably with the host language's syntax. They must be able to access information from the host language's type system, plugging in new types into the existing type-hierarchy by specifying new sub-typing relationships, new run-time type conversion mechanisms and by performing operator overloading. They must be able to add and retrieve elements from the environment, including elements that encapsulate new kinds of declaration information, in a way that avoids clashes with binding information from other extensions. Extensions must be able to easily construct valid tree values to specify translations to equivalent source code, or the generation of optimized code based on static compile-time analyses such as data-flow analyses. Generated compilers should provide appropriate errors in terms of the higher-level constructs used by the programmer, instead of reporting errors in terms of the generated host language translation (as with macros). This would shield programmers from having to analyse any generated code.

1.1.4.2 Non-Termination of Higher-Order Attribution

An issue when generating sound and terminating generated compilers in a higher-order attribute grammar framework such as Silver is the potential for improper termination of attribute evaluation during the execution of the generated compiler. As described in Section 2.3, in each evaluation step during attribute evaluation, an undefined evaluable attribute occurrence (i.e., an occurrence such that all the attribute instances its evaluation depends upon have been evaluated) is evaluated. This attribution process evaluates attribute instances one at a time until there are no undefined evaluable attribute instances. This may happen because

- all attribute instances have already been evaluated, in which case evaluation terminates normally, or
- a particular attribute instance has no associated attribute definition, or
- there is a circularity in the dependencies between two attribute instances, so that neither can be evaluated before the other.

Abnormal termination of evaluation results because there are no evaluable attribute instances even though there are undefined attribute instances, or because of function calls that terminate abnormally. Finally, attribute evaluation may not terminate because the total number of attribute instances may not be finite. Every time a new attributed syntax tree is constructed, new undefined attribute instances are added, possibly including instances that will cause the construction of more attributed syntax trees. Thus an infinite number of attributed syntax trees may be created. For example, the following is an attribute grammar (based on an example by Vogt *et al.* [17]) that is complete and has no circularities, but for which attribute evaluation does not terminate.

$$
\begin{aligned}
&p_R : R \ \longrightarrow \ X \\
&p_X : X \ \longrightarrow \\
&\quad l_A :: A = p_A() \\
&p_A : A \ \longrightarrow \\
&\quad l_X :: X = p_X()
\end{aligned}
$$

Here, l_A and l_X are local attributes (described in Section 2.2.1) that evaluate to new attributed trees on the nodes of the program syntax tree. The original program syntax tree $p_R(p_X())$ evaluates the local attribute l_A to a tree $p_A()$, which evaluates its local attribute l_X to a tree $p_X()$, which creates another local tree $p_A()$, and so on indefinitely.

Existing static analyses check higher-order attribute grammars for circularities and definedness [16, 17], but not for non-termination of tree creation. Thus it would be useful to develop static analyses that guarantee that tree creation for a given higher-order attribute grammar always terminates. Since the problem is undecidable, our challenge is to develop a sound and terminating procedure to detect higher-order termination for a class of useful grammars. Since modular static analyses guaranteeing termination of generated compilers are especially useful in our extensible language framework, the question of what modular guarantees could be given when composing a set of attribute grammars is also relevant.

1.2 Dissertation Contributions

1.2.1 Composable Specifications for Extension Semantics

When developing non-trivial extensions, there are questions as to how extensions may access and modify the host language's semantic analyses (such as types, environment and error-reporting) without interfering with other extensions doing the same. In this work, we show how to design host languages and language extensions for easy extensibility and safe composability so that the generated compilers act as expected. We examine the issue with an eye to the practical needs of extension developers and programmers. In Chapter 4, we present ableJ, an extensible language framework for Java 1.4 specified using Silver. It can be used to generate front-end translators from code written in Java extended with various features, to code in pure Java 1.4. The ABLEJ host language specification is designed for extensions that interact closely with the host language's type system and environment, and return error messages that reference extended code, rather than generated Java code. Extensions written to ABLEJ range from a simple macro-like enhanced `for` extension, to more interesting ones for complex number types, auto-boxing and algebraic data-types with pattern-matching.

It is necessary for the host language's type system and environment to be designed to allow for useful and composable extensions. One can develop a set of best practices for extension writers to follow that ensure that their extensions work with other well-behaved extensions. These best practices informally specify invariants on the composed language. An example of an invariant is a requirement that no extension type can be both a sub-type of a host language Java type, and a super-type of a host language Java type. This would ensure that there were no circularities in the extended type hierarchy when composing two or more independently developed extensions. ABLEJ's type system is described in more detail in Section 4.5.2.

In Chapter 4, we look at how Silver's features, such as aspect productions, collections and pattern-matching can be used to add new sub-typing relations, specify new type conversion mechanisms, and use the host environment in a composable fashion. Thus we show that it is possible to build high-level host specifications with provisions for extension writers to interact closely with the host environment and type systems, while at the same time avoiding problems when composed with other extensions. The generated compiler cannot always be fully validated, as in the situation where two extensions independently specify global transformations whose results depend on the order in which they are applied. Some impediments to composability are therefore beyond static specification analysis.

In Appendix A, we provide a second example of a language extension, that extends the Silver specification language with constructs to perform data-flow analysis during attribute evaluation. The extension allows extension writers to specify data-flow properties as temporal logic formulas which are checked during compilation (i.e., attribute evaluation) on models derived from the program's control-flow graphs. The model-checking results are obtained via calls to external model checkers and can be used within attribute definitions to generate optimized source code.

1.2.2 Termination Analysis for Higher-Order Attribute Evaluation

In Chapter 5, we present a static analysis on higher-order attribute grammars that detects non-terminating tree creation during attribute evaluation. The analysis provides extension writers and programmers with a guarantee that the generated compiler will not fail to terminate on account of improper attribute evaluation.

Vogt *et al.* [17] define well-defined higher-order attribute grammars as those that

1. are complete, in that every synthesized, inherited and local attribute instance on the nodes of any syntax tree has a definition,
2. are non-circular, in that there is no attribute instance whose values depends, either directly or indirectly, on itself, and
3. are such that only a finite number of new trees can ever be constructed during attribute evaluation.

Vogt *et al.* give extended versions of Knuth's completeness and circularity tests [16] to handle the first two conditions. Attribute evaluation for any grammar that passes these tests will not terminate abnormally. Thus an analysis that guarantees that no evaluation sequences with infinitely many tree creation steps exist, in conjunction with the completeness and circularity tests, suffices to ensure termination of higher-order attribute evaluation. The analysis presented here is the first, to our knowledge, that handles the issue of non-termination of tree creation, and ensure that a grammar satisfies the third condition above.

The analysis handles a specific class RHOAG of higher-order attribute grammars with attribute definition expressions that are a subset of the expressions for the general class of higher-order attribute grammars described in Chapter 2. Further, the non-termination problem is defined in the context of the evaluation model (described in Section 2.3) in which all attribute instances on a syntax tree will be evaluated. Finally, we assume that all grammars under consideration are complete and non-circular, with no non-terminating function calls. Under these assumptions, the problem of showing evaluation always terminates normally can be reduced to that of showing that no infinite sequences of local trees exist, in which each non-initial tree is created as a local attribute value on its predecessor.

The analysis first attempts to construct an ordering on the grammar's non-terminals, that ensures that in any tree creation sequence, there are only a finite number of trees created via inherited attribute accesses. It then constructs a set of rewrite rules, that if terminating, guarantee that no infinite tree creation sequences exist in which there are no trees created via inherited accesses. Therefore if both the non-terminal ordering and a set of terminating rules can be constructed, then no infinite tree creation sequences exist, and therefore attribute evaluation always terminates normally. Further, modular versions of these conditions can be defined, that if satisfied by an extension specification, ensure that any combination of that extension with other "well-behaved" extensions passes the monolithic test.

The analysis is simple enough to be easily proven correct and efficiently executed. But it is also useful enough to show termination for a large class of useful and interesting grammars. Most of the ableJ extension grammars are shown to be terminating by the monolithic termination analysis. Some also pass the modular version of the analysis.

1.3 Outline and Publications

1.3.1 Dissertation Outline

The main contributions of this dissertation are presented in Chapter 4 and Chapter 5.

- Chapter 2 gives background information on higher-order attribute grammars.
- Chapter 3 describes the Silver extensible language framework.
- Chapter 4 describes ableJ, an extensible host language specification for Java 1.4.
- Chapter 5 describes a static analysis on higher-order attribute grammars that detects non-terminating tree creation during attribute evaluation.
- Chapter 6 looks at related work in the area of extensible languages and attribute grammars, surveys future work, and concludes.
- Appendix A describes a second extension example in which the Silver specification language is extended with constructs to specify and perform data-flow analyses.
- Appendix B gives examples of higher-order attribute grammars, of tree creation during evaluation, and of rules generated by the termination analysis.
- Appendix C contains details of proofs related to the termination analysis.

1.3.2 Publication Details

The author acknowledges the assistance of other members of the MELT group to the success of his work.

- Research on the ableJ framework (described in Chapter 4) was published at the 21^{st} *European Conference on Object-Oriented Programming* (ECOOP) in 2007 [19]. The author was the main contributor to this work.
- Research on the termination analysis (described in Chapter 5) will be published at the 5^{th} *International Conference on Software Language Engineering* (SLE) in 2012 [20]. The author was the main contributor to this work.
- Research on the data-flow analysis extension to Silver (described in Appendix A) was published at the 5^{th} *International Workshop on Compiler Optimization Meets Compiler Verification* (COCV) in 2006 [21]. The author was the main contributor to this work.
- Finally, research on the design and development of the Silver tool (described in Chapter 3) was published at the 7^{th} *International Workshop on Language Descriptions, Tools and Applications* (LDTA) in 2007 [11]. While the author played a significant role in this work, his contribution was less than that of the other authors.

Chapter 2

Higher-Order Attribute Grammars

This chapter and Chapter 3 provide background material necessary for the descriptions in Chapters 4 and 5 of the contributions of this dissertation. This chapter gives an overview of higher-order attribute grammars (HOAG), focusing on the aspects that distinguish them from standard canonical attribute grammars as defined by Knuth [16]. In Chapter 3, we will describe how the various components of HOAGs are mapped onto the Silver specification language. These chapters provide background for the description in Chapter 4 of the process of using HOAGs for specifying composable language semantics.

This chapter also includes a formal description of higher-order attribute grammars and the concept of higher-order attribute evaluation. Much of the technical detail in this chapter, such as the notion of "normal termination" of attribute evaluation, and the notational conventions, are provided as background for the description in Chapter 5 of an analysis for termination of higher-order attribute evaluation.

Chapter Outline This chapter is outlined as follows:

- Section 2.1 gives preliminary definitions of context-free grammars, syntax trees and tree terms.
- Section 2.2 describes higher-order attribute grammars and the concept of attributed syntax trees, and provides formal definitions with examples.
- Section 2.3 defines the process of higher-order attribute evaluation as a sequence of evaluation steps in which undefined attribute occurrences are evaluated one at a time.

- NT = { `Expr` , `Term` , `Factor` , `Defs` }
- T = { `'let'` , `'in'` , `'+'` , `Id` , `IntConst` , `'='` , `';'` }
- S = `Expr`
- P :

```
Expr     ⟶  'let' Defs 'in' Expr | Term
Term     ⟶  Term '+' Factor | Factor
Factor   ⟶  Id | IntConst
Defs     ⟶  Id '=' Expr ';' Defs |
```

Figure 2.1: An example of a context-free grammar that defines a simple let expression language.

Readers familiar with higher-order attribute grammars may skip ahead, and refer back to explanations of individual terms or notations, as needed.

2.1 Preliminary Definitions

2.1.1 Context-Free Grammars

A context-free grammar (CFG) [12] may be defined formally as a quadruple $\langle\ NT,\ T,\ P,\ S\ \rangle$ where

- NT is a set of non-terminals.
- T is a set of terminals.
- P is a set of productions of the form $NT ::= (NT\ \cup\ T)*$.[1]

  $lhs : P \longrightarrow NT$ returns a production's left-hand side (LHS).

  $rhs : P \longrightarrow (NT\ \cup\ T)*$ returns a production's right-hand side (RHS).
- $S \in NT$ is the grammar's start symbol.

Figure 2.1 shows an example of a context-free grammar that defines a simple let expression language. An example of a syntactically valid program in this language is "`let x = 1; in x`".

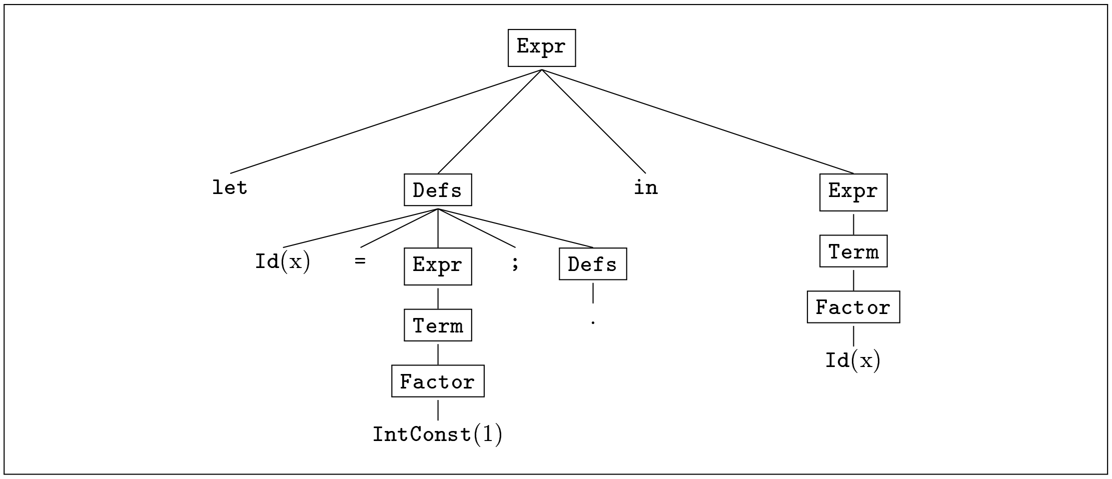


Figure 2.2: The syntax tree for the string "`let x = 1; in x`" for the grammar in Figure 2.1.

2.1.2 Syntax Trees

Figure 2.2 shows the syntax tree for the string "`let x = 1; in x`" for the grammar in Figure 2.1. Let N be a set of distinct and identifiable syntax tree nodes. We will reference parser generated syntax trees solely via their root nodes, as for example in the function *nodes* below. We will therefore not define a separate type for syntax trees (in addition to syntax tree nodes) to avoid confusion with the notion of tree terms defined below.

We have the following functions on syntax tree nodes:

- $\mathit{prod} : N \longrightarrow P$ returns a node's production.
- $\mathit{symbol} : N \longrightarrow NT \cup T$ returns a node's symbol.
- $\mathit{children} : N \longrightarrow (N)*$ returns a non-terminal node's children as a list of nodes. We will write $\mathit{child}(n, i)$ for n_i where $\mathit{children}(n) = n_1, ..., n_{n_p},\ 1 \leq i \leq n_p$.
- $\mathit{nodes} : N \longrightarrow \mathcal{P}(N)$ returns the set of nodes in a syntax tree, given its root node.[2]

[1] Here $X*$ represents the type "list of elements of type X".
[2] Here $\mathcal{P}(X)$ represents the powerset of the set X, i.e., the type "sets of sets of type X".

$$nodes(n) = \begin{cases} \{\ n\ \} & \text{if } symbol(n) \notin NT \\ \{\ n\ \} \cup nodes(n_1) \cup ... \cup nodes(n_{n_q}) & \text{if } symbol(n) \in NT, \\ & children(n) = n_1, ..., n_{n_q} \end{cases}$$

On any node n in a syntax tree, if $symbol(n) \in NT$ and $children(n) = n_1, ..., n_{n_q}$, then $lhs(prod(n)) = symbol(n)$ and $rhs(prod(n)) = symbol(n_1),\ ...,\ symbol(n_{n_q})$. Note that our definition of syntax trees includes all trees that may be derived from any non-terminal in the grammar. They do not necessarily have the start symbol S as their root node's symbol.

2.1.3 Tree Terms

We define an algebraic datatype *Term* for tree terms.

$$\begin{array}{llll} Term ::= & & & \\ & c & c \in T & \text{(terminal symbol)} \\ | & p\ (Term_1, ..., Term_{n_q}) & p \in P & \text{(production symbol and sub-trees)} \end{array}$$

Tree terms encapsulate the tree structure of a syntax tree but do not have syntax nodes (or attribute instances). This distinction between tree terms and syntax trees is needed for a couple of reasons. First, as described below, the process of attribute evaluation distinguishes between attributes whose evaluated values are tree terms, and those whose evaluated values are full-fledged syntax trees with attribute instances on their nodes.

Second, the termination analysis in Chapter 5 models attribute evaluation via rewrite sequences that rewrite tree terms. We define the following functions to perform conversions between tree terms and corresponding syntax trees.

- $term : N \longrightarrow Term$ returns the tree term for the sub-tree rooted at a given node.

$$term(n) = \begin{cases} symbol(n) & \text{if } symbol(n) \notin NT \\ q(term(n_1), ..., term(n_{n_q})) & \text{if } symbol(n) \in NT,\ prod(n) = q, \\ & children(n) = n_1, ..., n_{n_q} \end{cases}$$

- $newTree : Term \longrightarrow N$ given a tree term, returns the root node of a corresponding syntax tree.

2.2 Higher-Order Attribute Grammars

Attribute grammars [16] specify language semantics by associating typed values known as attributes to the nodes of each program syntax tree. The set of attribute instances on each node of such an attributed tree is specified by the attribute grammar, based on the node's non-terminal and its production. Each grammar production in the attribute grammar's context-free grammar has a set of functions, known as attribute definitions.

In the parser-generated tree, each syntax tree node has a set of attribute instances. Once the parser constructs a valid syntax tree, each node's attribute instances are computed by retrieving and evaluating the appropriate attribute definitions from its production and its parent production.

In higher-order attribute grammars , the set of possible attribute values is expanded from primitive values (such as integer values) to include higher-order tree values that are valid syntax sub-trees themselves [17, 22]. These are similar to the tree terms described in Section 2.1.3, and do not have attribute instances on them. Figure 2.3 gives the formal definition of higher-order attribute grammars. We now look at each of the components of this definition in turn, using an example (given in Figure 2.4) of a higher-order attribute grammar that computes the value of the let expressions defined by the grammar in Figure 2.1. The definitions in the `let` clauses are collected into an environment, which is used to perform look-ups in the `idRef` production.

2.2.1 The Components of Higher-Order Attribute Grammars

The Context-Free Grammar Attribute grammars define language semantics by specifying how attributes are to be computed on the nodes of syntax trees defined by context-free grammars. The formal definition in Figure 2.3 therefore includes a context-free grammar. The example in Figure 2.4 associates attribute definitions to the productions of the grammar in Figure 2.1. The grammar's productions are labeled with distinct identifiers that can be used within attribute definition expressions to construct tree values.

Primitive Types, Values and Functions Attribute definition expressions perform computations on the values of other attribute instances and on primitive values. In

An attribute grammar G is defined by the tuple
$\langle \langle NT, T, P, S \rangle, PT, PV, F, A_S, A_I, L, \mathit{defs} \rangle$ where

- $\langle NT, T, P, S \rangle$ is a context-free grammar.
- PT is a set of primitive types including `boolean`.
- PV is a set of primitive values including `true` and `false` of `boolean` type.
- F is a set of functions.

  $\mathit{inTypes} : F \longrightarrow (PT \cup NT \cup T)*$ returns a function's input types.
  $\mathit{outType} : F \longrightarrow (PT \cup NT \cup T)$ returns a function's output type.
  $\mathit{applyFun} : F \longrightarrow (PV \cup \mathit{Term})* \longrightarrow (PV \cup \mathit{Term})$ returns the result of applying a function to a list of values.

- A_S is a set of synthesized attributes.
- A_I is a set of inherited attributes.
- L is a set of higher-order local attributes.

  $\mathit{type}_a : (A_S \cup A_I \cup L) \longrightarrow (PT \cup NT \cup T)$ returns the type of a synthesized, inherited or local attribute.

- $@ \subseteq ((A_S \cup A_I) \times NT) \cup (L \times P)$ is the attribute occurrence relation.
- $\mathit{defs} : P \longrightarrow \mathcal{P}(\mathit{Defn})$ returns a production's attribute definition set.

Defn is the set of attribute definitions, each of which has the form below, where $e \in \mathit{Expr}$

	$\#0.a_S = e$	$a_S \in A_S$	(synthesized occurrence on parent)
$\mid$	$\#i.a_I = e$	$a_I \in A_I$	(inherited occurrence on child)
$\mid$	$l = e$	$l \in L$	(local occurrence)
$\mid$	$l.a_I = e$	$l \in L,\ a_I \in A_I$	(inherited occurrence on local)

Expr is the set of attribute definition expressions, each of which has the form below, where $e_i \in \mathit{Expr}$

	$\#i$		(parent or child tree)
$\mid$	$\#i.a$	$a \in (A_S \cup A_I)$	(attribute occurrence)
$\mid$	$l.a$	$l \in L,\ a \in (A_S \cup A_I)$	(attr. occurrence on local)
$\mid$	$q(e_1, \ldots, e_{n_q})$	$q \in P$	(tree creation)
$\mid$	c	$c \in T$	(terminal symbol)
$\mid$	$f(e_1, \ldots, e_{n_f})$	$f \in F$	(function call)
$\mid$	`if` e_C `then` e_T `else` e_E		(conditional expression)

Figure 2.3: A formal definition for higher-order attribute grammars.

- $NT = \{$ Expr, Term, Factor, Defs, Env $\}$ - $S =$ Expr
- $T = \{$ 'let', 'in', '+', Id, IntConst, '=', ';' $\}$
- $PT = \{$ integer, boolean, string $\}$ - $PV = \{$ true, false, -1 $\}$
- $F = \{$ toInt :: string $\longrightarrow$ integer, + :: (integer $\longrightarrow$ integer) $\longrightarrow$ integer,
  == :: (string $\longrightarrow$ string) $\longrightarrow$ boolean, || :: (boolean $\longrightarrow$ boolean) $\longrightarrow$ boolean $\}$
- $A_S = \{$ defs :: Env, value :: integer, found :: boolean $\}$
- $A_I = \{$ env :: Env, lookFor :: string $\}$
- $L = \{$ lenv :: Env $\}$
- @ $= \{$ ⟨defs, Defs⟩, ⟨value, Expr⟩, ⟨value, Term⟩, ⟨value, Factor⟩, ⟨value, Env⟩,
  ⟨env, Expr⟩, ⟨env, Term⟩, ⟨env, Factor⟩, ⟨lookFor, Env⟩, ⟨found, Env⟩, ⟨lenv, idRef⟩ $\}$
- $P = \{$ let, exprTerm, add, termFactor, idRef, intConst, consDefs, emptyDefs,
  consEnv, emptyEnv $\}$

```
let :: Expr ⟶ 'let' Defs 'in' Expr
      #0.value = #4.value
      #4.env = #2.defs
exprTerm :: Expr ⟶ Term
      #0.value = #1.value
      #1.env = #0.env
add :: Term ⟶ Term '+' Factor
      #0.value = #1.value + #3.value
      #1.env = #0.env
      #3.env = #0.env
termFactor :: Term ⟶ Factor
      #0.value = #1.value
      #1.env = #0.env
idRef :: Factor ⟶ Id
      lenv = #0.env
      lenv.lookFor = #1.lexeme
      #0.value = if lenv.found then lenv.value else −1
intConst :: Factor ⟶ IntConst
      #0.value = toInt (#1.lexeme)

consDefs :: Defs ⟶ Id '=' Expr ';' Defs
      #0.defs = consEnv (#1, intConst (IntConst (#3.value)), #5.defs)
emptyDefs :: Defs ⟶ ';'
      #0.defs = emptyEnv ()
consEnv :: Env ⟶ Id Factor Env
      #0.found = #1.lexeme == #0.lookFor || #3.found
      #0.value = if #1.lexeme == #0.lookFor then #2.value else #3.value
      #3.lookFor = #0.lookFor
emptyEnv :: Env ⟶
      #0.found = false
```

Figure 2.4: An example of the definition of a higher-order attribute grammar.

Figure 2.3, PT is a set of primitive types. In Figure 2.4, these are the integer, boolean and string types. PV is a set of primitive values, with types in PT. In the simple example in Figure 2.4, there are only three primitive values. Finally, F is a set of functions over the values in PV. Figure 2.4 includes each primitive function's type signature.

Synthesized Attributes In Figure 2.3, A_S is a set of synthesized attributes. Synthesized attributes implement semantic analyses which require information from sub-trees to be collected, passed up, and made available elsewhere in the program. An example of an analysis commonly implemented using synthesized attributes is the construction of symbol tables in which information about variable bindings is gathered up from all the variable declarations in a particular sub-tree. In the example in Figure 2.4, the synthesized attribute `defs` does exactly this. Other examples of values computed using synthesized attributes are generated code and error messages. The synthesized attribute instances on a node are defined on its production.

Inherited Attributes In Figure 2.3, A_I is a set of inherited attributes. Inherited attributes are used when information needs to be passed down the syntax tree. An example would be the program environment passed down the tree for binding analysis. In the example in Figure 2.4, `env` is an example of such an inherited attribute. The inherited attribute occurrences on a non-terminal may be defined by any production with that non-terminal on its right-hand side. Inherited attribute instances on a node are therefore defined by its parent's production. A production may also set the values of inherited attribute instances on the roots of its local attributes.

Local Attributes In Figure 2.3, L is a set of higher-order local attributes. Local attributes are associated with productions, and not with non-terminals. A production defines the tree values of its local attributes. During attribute evaluation, the syntax tree corresponding to a local attribute's evaluated tree value is created as a local attribute instance on the production's node. We will refer to the node on which a local attribute instance is created as the local attribute's parent node. Like regular parent nodes, a local attribute's parent node sets the inherited values on its root, and accesses synthesized values off its root. A production can thus use its local attributes to perform side-computations such as symbol look-ups during the evaluation of its other definitions. The example grammar in Figure 2.4 has one local attribute `lenv`. This is used by the `idRef` production to look up its identifier's lexeme in its environment.

The Occurrence Relation In Figure 2.3, @ specifies the occurs-on or occurrence relation of the grammar's attributes. It specifies which attributes decorate which non-terminals (in the case of synthesized and inherited attributes), or productions (in the case of local attributes). Each syntax tree node has an attribute instance for each synthesized and inherited attribute that occurs on its non-terminal symbol, and for each local attribute that occurs on its production. In the example in Figure 2.4, the inherited attribute `env` decorates the `Expr`, `Term` and `Factor` non-terminals, while the local attribute `lenv` decorates the `idRef` production.

Attribute Definitions Finally, $\mathit{defs}(p)$ returns the set of attribute definitions associated with each grammar production p. Each definition specifies the function that computes the value of a synthesized, inherited or local attribute instance on either the production's node, on a child node, or on the root node of a locally created tree. Definitions are of the form *occurrence* = *expression*. Their syntax is specified in Figure 2.3. Information flow between tree nodes results from defining attributes in terms of the values of attribute occurrences on neighboring nodes. There are restrictions on which attribute occurrences may be defined on a given production. Each definition on a production may define one of the following four kinds of attribute occurrences:

- synthesized attribute occurrences on the production's node,
- local attribute occurrences defined on the production's node,
- inherited attribute occurrences on the production's node's children, and
- inherited attribute occurrences on the root nodes of the production's node's local attribute instance values.

Attribute Definition Expressions The right-hand sides of definitions are attribute defining expressions of type *Expr*. The syntax of expressions may be different in different higher-order attribute grammar frameworks. Here, we define a simple class of expressions for the purpose of describing the process of attribute evaluation in the next section. Actual frameworks such as Silver (described in Chapter 3) may provide other kinds of definition expressions.

In Figure 2.3, expressions are constructed using:

- references to trees in the production signature,
- accesses to attribute occurrences on the node or its children,
- accesses to attribute occurrences on the roots of local trees,
- explicit tree creation using production names and tree-valued arguments,
- explicit tree creation using terminal symbols,
- function calls, and
- conditional expressions.

Note that the set *Expr* of expressions includes all tree terms, i.e., $\mathit{Term} \subset \mathit{Expr}$.

2.2.2 Type-Correct Attribute Grammars

Attribute definitions must be correctly typed. The types of expressions on the right-hand sides of attribute definitions (defined in Figure 2.5) must be the same as those of the attribute occurrences they define. Note that the static type and validity of an expression or definition are determined with respect to a specific production, since signature variables and local attribute references must be resolved for a particular production. We will assume that we are analysing only type-correct grammars.

2.2.3 Composing Higher-Order Attribute Grammars

In higher-order attribute grammar based extensible language frameworks, multiple attribute grammar specifications are composed to generate a single type-correct attribute grammar. In most cases, there is a host language attribute grammar, and extension attribute grammars that specify extra constructs and add other definitions to the host language grammar. Given a host language attribute grammar

$$G_H = \langle\ \langle\ NT_H,\ T_H,\ P_H,\ S\ \rangle,\ PT,\ PV,\ F,\ A_S,\ A_I,\ L_H,\ \mathit{defs}_H\ \rangle$$

$type_e : P \longrightarrow Expr \longrightarrow (PT \ \cup \ NT \ \cup \ T)$ returns an expression's type.

For an expression e on a production p, $type_e(p,\ e)$ is defined as follows (here $|X|^i$ represents the $i^{\texttt{th}}$ element of the list X):

$$\frac{}{type_e(p,\ \#0) = lhs(p)} \qquad \text{(type of parent)}$$

$$\frac{1 \leq i \leq |rhs(p)|}{type_e(p,\ \#i) = |rhs(p)|^i} \qquad \text{(type of } i^{\texttt{th}} \text{ child)}$$

$$\frac{a@lhs(p)}{type_e(p,\ \#0.a) = type_a(a)} \qquad \text{(attr. occ. on parent)}$$

$$\frac{1 \leq i \leq |rhs(p)|,\ \ a@(|rhs(p)|^i)}{type_e(p,\ \#i.a) = type_a(a)} \qquad \text{(attr. occ. on child)}$$

$$\frac{l@p,\ \ a@type_a(l)}{type_e(p,\ l.a) = type_a(a)} \qquad \text{(attr. occ. on local)}$$

$$\frac{rhs(q) = type_e(p, e_1), ...,\ type_e(p, e_{n_q})}{type_e(p,\ q(e_1, ...,\ e_{n_q})) = lhs(q)} \qquad \text{(tree creation)}$$

$$\frac{}{type_e(p,\ c) = c} \qquad \text{(terminal symbol)}$$

$$\frac{inTypes(f) = type_e(p, e_1), ...,\ type_e(p, e_{n_f})}{type_e(p,\ f(e_1, ...,\ e_{n_f})) = outType(f)} \qquad \text{(function call)}$$

$$\frac{type_e(p, e_C) = \texttt{boolean},\ type_e(p, e_T) = type_e(p, e_E)}{type_e(p,\ \texttt{if}\ e_C\ \texttt{then}\ e_T\ \texttt{else}\ e_E) = type_e(p,\ e_T)} \qquad \text{(conditional expression)}$$

We have the following conditions on attribute definitions.

- $\#0.a_S = e :\ \ a_S@lhs(p),\ type_e(p,\ e) = type_a(a_S)$
- $\#i.a_I = e :\ \ 1 \leq i \leq |rhs(p)|,\ a_I@(|rhs(p)|^i),\ type_e(p,\ e) = type_a(a_I)$
- $l = e :\ \ l@p,\ type_e(p,\ e) = type_a(l)$
- $l.a_I = e :\ \ l@p,\ a_I@type_a(l),\ type_e(p,\ e) = type_a(a_I)$

Figure 2.5: The types of attribute definition expressions.

the attribute grammar for an extended version of this host language would be given by

$$G_{H+E} = \langle\ \langle\ NT_H\ \cup\ NT_E,\ T_H\ \cup\ T_E,\ P_H\ \cup\ P_E,\ S\ \rangle,\ PT,\ PV,\ F,$$

$$A_{SH}\ \cup\ A_{SE},\ A_{IH}\ \cup\ A_{IE},\ L_H\ \cup\ L_E,\ \mathit{defs}_H\ \cup\ \mathit{defs}_E\ \rangle$$

Note that the extended language has the same start symbol (S), primitive types (PT), primitive values (PV) and functions (F) as the host language. The extended grammar may define new non-terminals, terminals, productions, synthesized attributes, inherited attributes, local attributes, as well as new definitions, both to new and existing productions.

2.3 Higher-Order Attribute Evaluation

2.3.1 Attributed Syntax Trees

Each node of a parser-generated syntax tree has a set of attribute instances, where each attribute instance is a slot for the attribute value that will be computed during the process of attribute evaluation. Each non-terminal node has an attribute instance for every synthesized and inherited attribute decorating its non-terminal, and every local occurring on its production. Finally, every syntax tree leaf node that is a terminal has a default attribute, `lexeme`, which is set by the parser to the string value of its token. All other attribute instances on the tree are initially undefined.

Figure 2.6 shows the initial, unevaluated attributed version of the syntax tree in Figure 2.2 for the string "`let x = 1; in x`". Each `Defs` node has a single attribute instance, for the synthesized attribute `defs`. Each `Expr`, `Term` and `Factor` node has at least two attribute instances, for the synthesized attribute `value` and the inherited attribute `env`, shown in this order. Note that the `Factor` node constructed with the `idRef` production has an extra attribute instance, for the local attribute `lenv`.

Each attribute instance is defined by its node and attribute. We represent attribute instances with the type

$$\mathit{Instance} \equiv N \times (A_S\ \cup\ A_I\ \cup\ L)$$

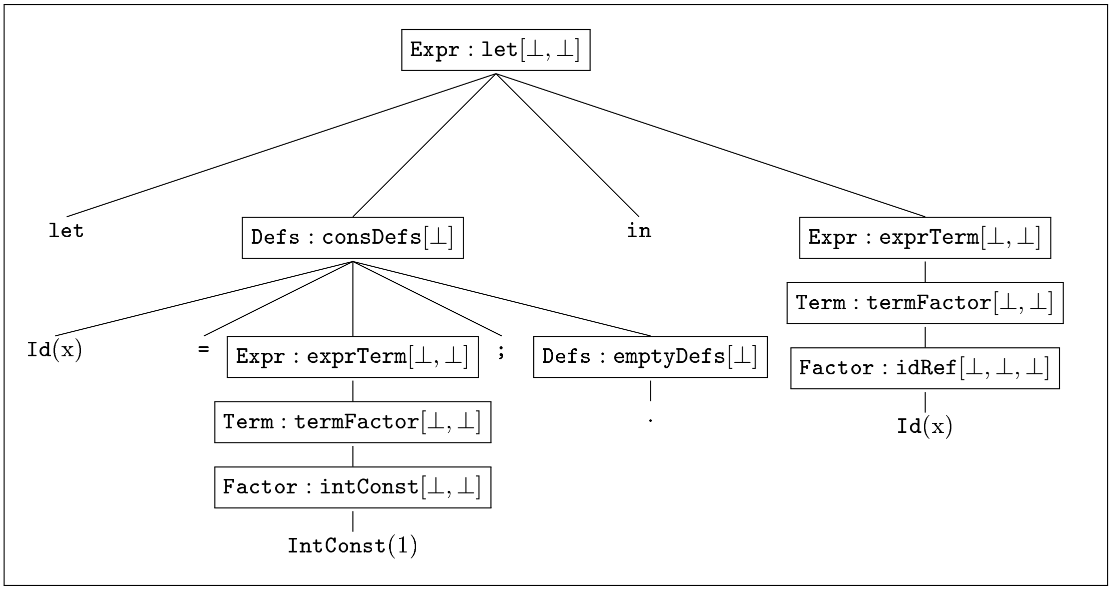


Figure 2.6: The unevaluated attributed syntax tree for "`let x = 1; in x`".

We will write $n\#a$ for an attribute instance of a on a node n. The function $instances$: $N \longrightarrow \mathcal{P}(Instance)$ returns all the attribute instances in a tree given its root node.

$$instances(n) = \begin{cases} \{n\#\texttt{lexeme}\} & \text{if } symbol(n) \in T \\ \{n\#a | a@symbol(n)\} \cup \{n\#a | a@q\} & \text{if } symbol(n) \in NT, prod(n) = q, \\ \quad \cup\ instances(n_1) \cup ... \cup\ instances(n_{n_q}) & \quad children(n) = n_1, ..., n_{n_q} \end{cases}$$

An attribution is a map that associates values to the attribute instances of a set of syntax trees, where each value is of the right type and consistent with its attribute definition. The set of possible values assigned by an attribution includes

- primitive values in the set PV,
- tree terms in the set $Term$,
- pointers to the roots of attributed syntax trees, represented by the nodes in the set N, and
- the special value $\perp$ that represents undefined attribute instances.

We therefore represent attributions with the type

$$Attribution \equiv Instance \longrightarrow (PV \ \cup \ Term \ \cup \ N \ \cup \ \{\perp\})$$

2.3.2 Attribute Evaluation

Attribute evaluation refers to the process by which an attribution is computed for an unevaluated attributed tree (i.e., one in which all attribute instances are undefined). The process starts with a syntax tree, usually constructed by a parser, with all of its attribute instances undefined, and proceeds as a series of steps in which attribute instances are evaluated one at a time. Each evaluation step results in an evaluation state, which encapsulates information about what trees are being attributed, what attribute instances remain to be evaluated and the values of attribute instances that have been defined. We will refer to this sequence of evaluation states as an evaluation sequence.

Evaluation State A state is fully defined by the nodes and edges that define its syntax trees, and an attribution to the attribute instances on the nodes of the trees. For clarity, we will represent each evaluation state as a pair $\langle \mathcal{T},\ \Gamma \rangle$ of the root nodes of its trees, and its attribution. We assume that other nodes and edges are available via global functions such as *children*. We therefore represent evaluation states with the type

$$State \equiv \mathcal{P}(N) \ \times \ Attribution$$

Evaluable Attribute Instance An attribute instance is evaluable in an evaluation state if all the attribute instances needed to compute its value have been evaluated. In each step of the evaluation sequence, an undefined evaluable attribute instance on one of the trees is selected and set to the evaluated value of its attribute defining expression. The first step of the evaluation sequence must therefore evaluate an attribute whose definition is a constant, or depends solely on parser-set attributes such as `lexeme`.

The attribute evaluation process evaluates attribute instances one at a time until there are no more undefined evaluable attribute instances. This may happen because

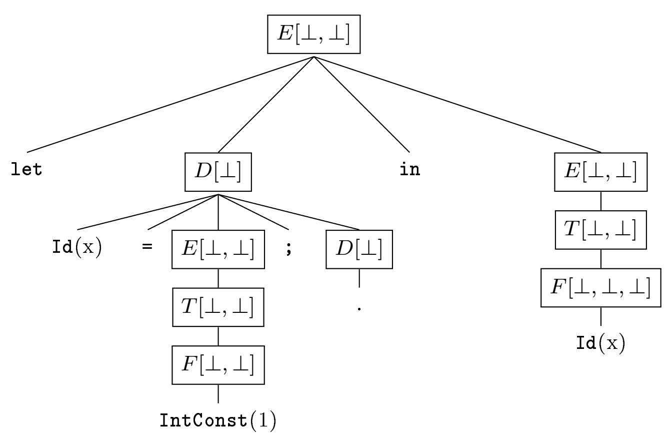
- all attribute instances have already been evaluated, or
- a particular attribute instance has no associated attribute definition, or

- there is a circularity in the dependencies between two attribute instances, so that neither can be evaluated before the other.

Abnormal termination of evaluation results because there are no evaluable attribute instances even though there are undefined attribute instances, or because of function calls that terminate abnormally.

The semantics of local attribute evaluation steps are different from those of other kinds of higher-order attribute evaluation steps (i.e., those evaluating synthesized or inherited attribute occurrences). The latter result only in tree terms. When a local attribute instance is evaluated, on the other hand, the computed tree value is converted into a full-fledged syntax tree with its own (undefined) attribute instances. This new syntax tree is added to the set of trees that define each evaluation state. Later evaluation steps will evaluate the attribute instances on this new tree. Note that the syntax of a higher-order expression does not indicate whether it will be evaluated to a tree term or syntax tree. This is decided by the evaluation context, i.e., whether a local attribute instance is being evaluated. An informal description of the process of attribute evaluation process for higher-order attribute grammars is given in Figure 2.7.

An example of the attribution of a tree is shown below. We start with the unevaluated tree for the string "`let x = 1; in x`". To conserve space, we have omitted production names and abbreviated non-terminal names. The full tree is shown in Figure 2.6.



The sequence evaluates the synthesized attribute `value` on the root `Expr` node. It is one of multiple possible evaluation sequences. The first steps of the sequence evaluate

Attribution of the unevaluated tree t proceeds as follows:

- **SET** T to $\{t\}$.

- **WHILE** there is an evaluable attribute occurrence $n\#a$ in a tree in T

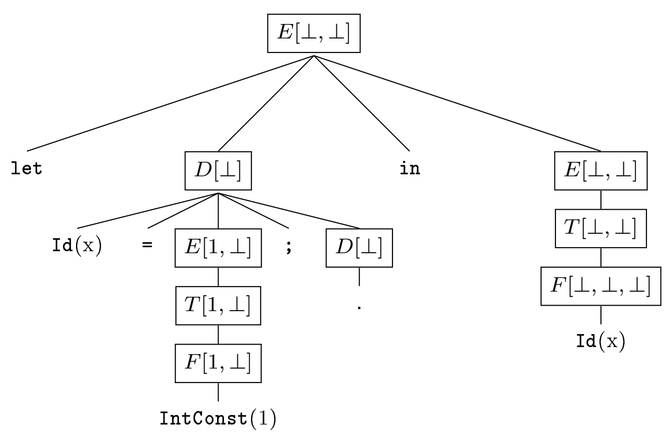
  - **IF**a is a synthesized or local attribute
    - **SET** e to $n\#a$'s defining expression e, specified in n's production.

    ELSE
    - **SET** e to $n\#a$'s defining expression e, specified in n's parent node's production.
  - **SET** v to the evaluated value of e.
  - **IF** a is a local attribute
    - **ADD** an attributed version of the tree term v to T.
    - **SET** $n\#a$ to the root of this tree.

    ELSE
    - **SET** $n\#a$ to v.

Figure 2.7: An outline of the process of higher-order attribute evaluation.

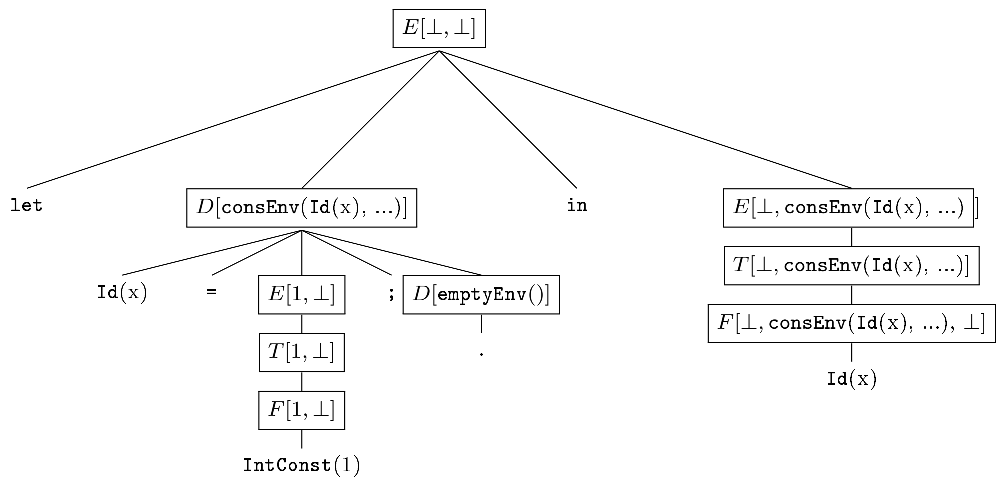
the `value` attribute instances on the nodes of the integer constant expression sub-tree.



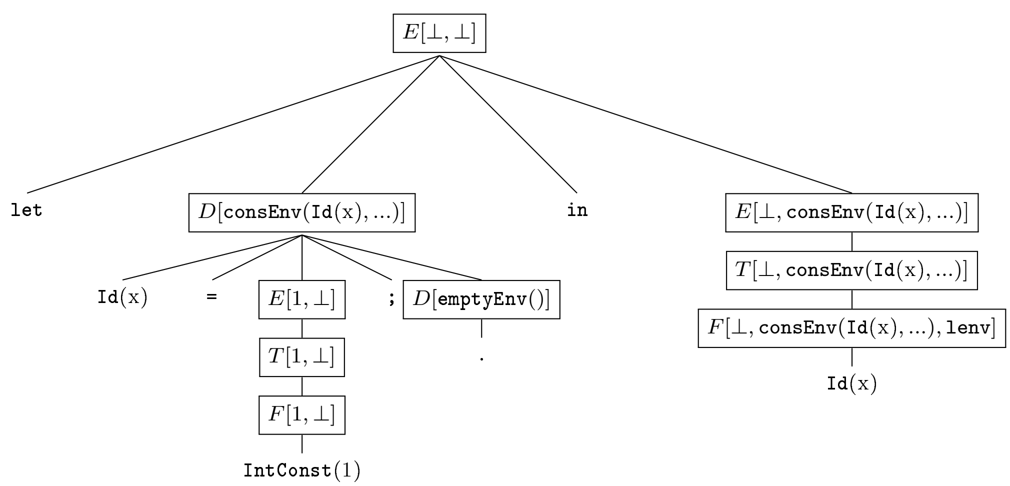
Once the `value` attribute instances have been evaluated, the value of the `defs` attribute instances are computed on the `Defs` nodes. The `let` expression's definitions are used to construct a tree term of the form

$$\texttt{consEnv}\ (\texttt{Id}(\mathrm{x}),\ \texttt{intConst}\ (\texttt{IntConst}(1)),\ \texttt{emptyEnv}\ ())$$

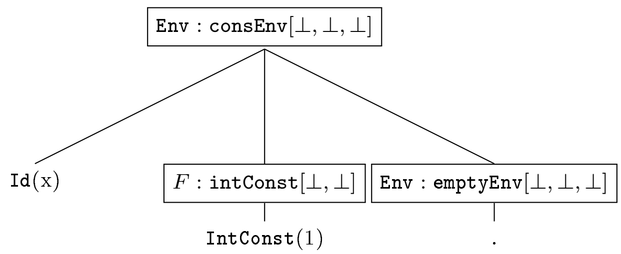
The definitions are used to compute the environment of the let expression. The values of the `defs` and `env` attribute instances are tree terms, whose evaluation does not result in the creation of new attributed syntax trees.



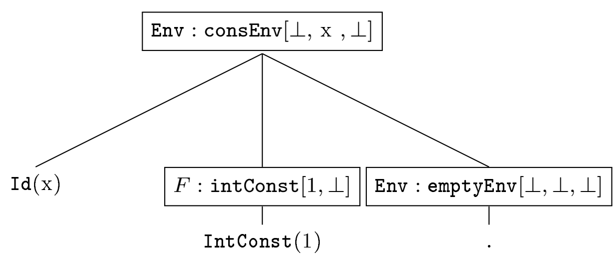
A new attributed tree is created when the `idRef` production's local attribute `lenv` is evaluated to the value of its environment.



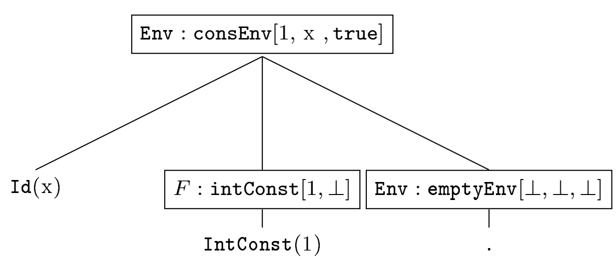
All of the new tree's attribute instances are undefined. Each `Env` node has three attributes - `value`, `lookFor` and `found` - shown in that order.



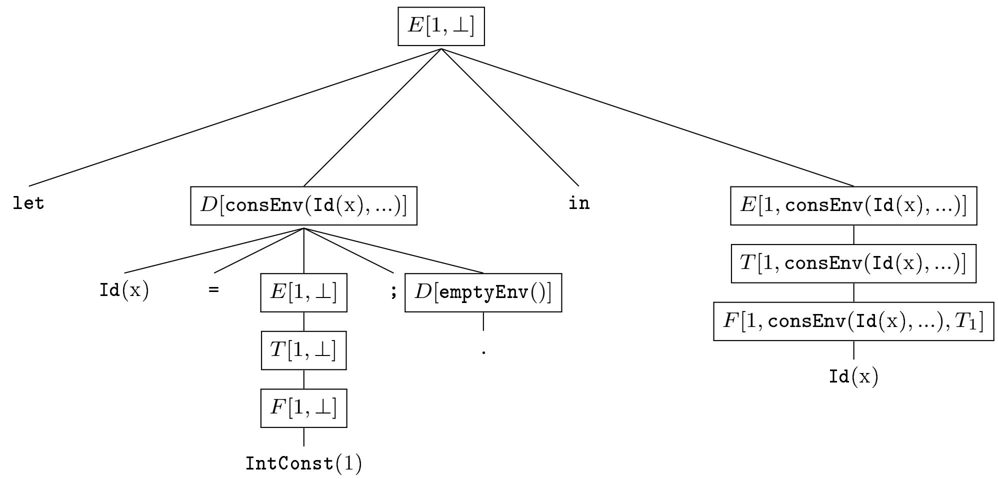
In the next few steps, the inherited attribute `lookFor` on the root node of the new tree is evaluated (using a definition retrieved from the `idRef` production).



A few steps later, the values of synthesized attributes on the root of the local tree are available for access.



The `idRef` node accesses the `value` attribute off the root of the local, and passes it up to the root of the original tree.



Further examples of higher-order attribute evaluation for sample grammars are given in Appendix B.1.1 and Appendix B.2.1.

Different attribute evaluation frameworks perform evaluation in different ways. Most, such as Silver [11] and JastAdd [10], evaluate only the attributes required to compute the values of desired synthesized attributes on the root node. Thus in the example above, the `env` attributes on the integer constant nodes would not be evaluated because they are never used. Similarly the root's `env` attribute is also never used. If it were queried for some reason, this would result in the evaluation sequence terminating abnormally.

In our formal definition, we assume a simple model of evaluation that evaluates all attribute instances. This necessitates the assumption of definedness and the absence of any inherited attributes on the root node. The latter can be achieved for this example by adding a root non-terminal that passes an empty environment to the let expression.

Evaluation Sequence The evaluation of a given tree can be formally described as a sequence of evaluation states generated via two functions: *initState* and *nextState*.

- $\mathit{initState} : \mathit{Term} \longrightarrow \mathit{State}$ returns the initial evaluation state for a given tree term. It consists of a single syntax tree, and an attribution in which all the tree's attribute instances are undefined.
  $\mathit{initState}(t) = \langle \{n\},\ \{n'\#a' \mapsto \perp \mid n'\#a' \in \mathit{instances}(n)\} \rangle$
  where $n = \mathit{newTree}(t)$

- *nextState* : $State \longrightarrow Instance \longrightarrow State$ defines the non-initial states in an evaluation sequence. Given an evaluation state $\langle \mathcal{T},\ \Gamma\rangle$ and a specific undefined evaluable attribute instance $n\#a \in instances(t)$ where $t \in \mathcal{T}$, $nextState(\langle \mathcal{T},\ \Gamma\rangle, n\#a)$ is the state that results from evaluating $n\#a$ in $\langle \mathcal{T},\ \Gamma\rangle$.

We write $\langle \mathcal{T}_{i+1},\ \Gamma_{i+1}\rangle = nextState(\langle \mathcal{T}_i,\ \Gamma_i\rangle, n_i\#a_i)$ as $\langle \mathcal{T}_i,\ \Gamma_i\rangle \overset{n_i\#a_i}{\rhd} \langle \mathcal{T}_{i+1},\ \Gamma_{i+1}\rangle$. An evaluation sequence for a tree t_0 is therefore given by a sequence

$$\langle \mathcal{T}_0,\ \Gamma_0\rangle \overset{n_0\#a_0}{\rhd} \langle \mathcal{T}_1,\ \Gamma_1\rangle \overset{n_1\#a_1}{\rhd} \langle \mathcal{T}_2,\ \Gamma_2\rangle \overset{n_2\#a_2}{\rhd} \ldots$$

where $\langle \mathcal{T}_0,\ \Gamma_0\rangle = initState(t_0)$ and $n_i\#a_i$ is an undefined evaluable attribute instance in the state $\langle \mathcal{T}_i,\ \Gamma_i\rangle$. Since there may be multiple attribute instances that can be evaluated in a given state, *initState* and *nextState* define multiple evaluation sequences, based on which undefined evaluable attribute instance is selected for evaluation in a given state.

The function *dep* returns the set of attribute instances required to evaluate a particular expression at a given point during evaluation. It is defined in Figure 2.8. An attribute instance $n\#a$ with defining expression e is evaluable in a state Γ if e is evaluable, i.e., if

$$\forall (n'\#a') \in dep(n,\ e,\ \Gamma)\ .\ \Gamma[n'\#a'] \neq \bot$$

The set of attribute occurrences required for the evaluation of an expression can only be computed lazily at run-time, because of the semantics of conditional expressions and local attributes. The function *eval* returns an evaluable expression's evaluated value and is defined in Figure 2.9. Attribute accesses, function calls and tree-creating steps are all assumed to terminate atomically with valid values. Conditional expressions are evaluated lazily based on the value of the condition.

Each non-initial evaluation step is generated from a state $\langle \mathcal{T},\ \Gamma\rangle$ based on the kind of the attribute occurrence $n\#a$ selected for evaluation.

- If $a \in A_S$ and $(\#0.a = e) \in defs(prod(n))$,
  then $\langle \mathcal{T},\ \Gamma\rangle \overset{n\#a}{\rhd} \langle \mathcal{T}, \Gamma[n\#a \mapsto eval(n,\ e,\ \Gamma)]\rangle$.
- If $a \in A_I,\ n = child(n_p, i)$ and $(\#i.a = e) \in defs(prod(n_p))$,
  then $\langle \mathcal{T},\ \Gamma\rangle \overset{n\#a}{\rhd} \langle \mathcal{T}, \Gamma[n\#a \mapsto eval(n_p,\ e,\ \Gamma)]\rangle$.
- If $a \in L,\ (a = e) \in defs(prod(n))$ and $t_L = newTree(eval(n,\ e,\ \Gamma))$, then

$dep : N \longrightarrow Expr \longrightarrow Attribution \longrightarrow \mathcal{P}(Instance)$ returns the set of attribute instances required to evaluate an expression on a given node during evaluation.

For an expression e on a node n, $dep(n,\ e,\ \Gamma)$ is defined as follows:

$dep(n,\ \#i,\ \Gamma) = \{\ \}$	(parent or child tree)
$dep(n,\ \#0.a,\ \Gamma) = \{n\#a\}$	(instance on parent)
$dep(n,\ \#i.a,\ \Gamma) = \{child(n,i)\#a\}$	(instance on child)
$dep(n,\ l.a,\ \Gamma) = \begin{cases} \{n\#l\} & \text{if } \Gamma[n\#l] = \bot \\ \{(\Gamma[n\#l])\#a\} & \text{otherwise} \end{cases}$	(local attr. instance if undefined, else the instance on the local)
$dep(n, q(e_1, ..., e_{n_q}),\ \Gamma) = \bigcup_{i=1}^{n_q} dep(n,\ e_i,\ \Gamma)$	(req. instances on all sub-exprs.)
$dep(n,\ c,\ \Gamma) = \{\ \}$	(terminal symbol)
$dep(n, f(e_1, ..., e_{n_f}),\ \Gamma) = \bigcup_{i=1}^{n_f} dep(n,\ e_i,\ \Gamma)$	(req. instances on all sub-exprs.)
$dep(n,\ \texttt{if}\ e_C\ \texttt{then}\ e_T\ \texttt{else}\ e_E,\ \Gamma) = \begin{cases} dep(n, e_C, \Gamma) & \text{if } dep(n, e_C, \Gamma) \neq \bot \\ dep(n, e_T, \Gamma) & \text{if } eval(n, e_C, \Gamma) = \texttt{true} \\ dep(n, e_E, \Gamma) & \text{otherwise} \end{cases}$	(cond.'s req. inst. if not evaluable else req. instances in sub-expr.)

Figure 2.8: The set of attribute instances required to evaluate an expression.

$eval : N \longrightarrow Expr \longrightarrow Attribution \longrightarrow (PV \ \cup \ Term)$ returns the value of an evaluable expression evaluated on a given node for a given attribution.

For an expression e on a node n where $\forall(n'\#a') \in dep(n,\ e,\ \Gamma)\ .\ \Gamma[n'\#a'] \neq \bot$, $eval(n,\ e,\ \Gamma)$ is defined as follows:

$$
\begin{array}{ll}
eval(n,\ \#i,\ \Gamma) = term(child(n,i)) & (i^{\texttt{th}} \text{ signature tree}) \\
eval(n,\ \#0.a,\ \Gamma) = \Gamma[n\#a] & (\text{instance value on parent}) \\
eval(n,\ \#i.a,\ \Gamma) = \Gamma[child(n,i)\#a] & (\text{instance value on child}) \\
eval(n,\ l.a,\ \Gamma) = \Gamma[(\Gamma[n\#l])\#a] & (\text{instance value on local}) \\
 & \\
eval(n,\ q(e_1, ...,\ e_{n_q}),\ \Gamma) & (\text{evaluated tree term}) \\
\quad = q(eval(n,\ e_1,\ \Gamma), ...,\ eval(n,\ e_{n_q},\ \Gamma)) & \\
 & \\
eval(n,\ c,\ \Gamma) = c & (\text{terminal symbol}) \\
 & \\
eval(n,\ f(e_1, ...,\ e_{n_f}),\ \Gamma) & (\text{value of function call}) \\
\quad = applyFun(f,\ eval(n, e_1, \Gamma), ..., eval(n, e_{n_f}, \Gamma)) & \\
 & \\
eval(n,\ \texttt{if}\ e_C\ \texttt{then}\ e_T\ \texttt{else}\ e_E,\ \Gamma) & \\
\quad = \begin{cases} eval(n,\ e_T,\ \Gamma) & \text{if } eval(n,\ e_C,\ \Gamma) = \texttt{true} \\ eval(n,\ e_E,\ \Gamma) & \text{otherwise} \end{cases} & (\text{evaluated sub-expression})
\end{array}
$$

Figure 2.9: Defining the evaluated value of an attribute definition expression.

$\langle \mathcal{T},\ \Gamma\rangle \overset{n\#a}{\triangleright} \langle \mathcal{T} \cup\ \{t_L\},\ \Gamma[n\#a \mapsto t_L]\ \cup\ \{[n'\#a' \mapsto \bot] \mid n'\#a' \in \mathit{instances}(t_L)\}\rangle.$

- If $a \in A_I,\ n = \Gamma[n_p\#a]$ and $(l.a = e) \in \mathit{defs}(\mathit{prod}(n_p))$,
  then $\langle \mathcal{T},\ \Gamma\rangle \overset{n\#a}{\triangleright} \langle \mathcal{T}, \Gamma[n\#a \mapsto \mathit{eval}(n_p,\ e,\ \Gamma)]\rangle$.

We will refer to the evaluation steps of the third kind as tree creation steps.

2.3.3 Normal and Abnormal Termination, and Non-Termination

The process of attribute evaluation described above evaluates the attribute instances one at a time until there are no undefined evaluable attribute instances. This may happen because all attribute instances have been evaluated, in which case evaluation is said to have terminated normally. Abnormal termination of evaluation results because of circularities in the attribute definitions, or because of attribute instances with no associated definitions, or because of function calls that terminate abnormally.

If the grammar has passed the higher-order circularity test, then in every evaluation state an order exists in which to evaluate all undefined attribute instances, so that an attribute instance is evaluated only after all its required attribute instances have been evaluated. If the grammar has passed the higher-order definedness test, then any attribute instance selected for evaluation has an associated definition. If a grammar passes the circularity test, passes the definedness test and is such that all functions terminate normally, then its evaluation will not terminate abnormally. In other words, for such a grammar, if evaluation terminates, then all attribute instances have been evaluated. When analysing grammars, we assume that all function calls terminate normally.

Finally, there is the possibility that attribute evaluation may not terminate because the total number of attribute instances is not finite. Every time a local attribute instance is evaluated, a new syntax tree with its own undefined attribute instances is created. These new instances may possibly include new local attribute instances. Thus an infinite number of local trees may be created. This is true even for non-circular, complete grammars with functions that terminate normally. In Chapter 5, we present an analysis to statically analyse such higher-order attribute grammars for non-terminating local tree creation.

Chapter 3

Silver

This chapter provides background information on the Silver extensible language framework, focusing on how host language and extension syntactic and semantic specifications are written in Silver's specification language. As described in Section 1.1.3, Silver is a higher-order attribute grammar tool with which modular language specifications can be written and automatically composed to generate a compiler for an extended version of a particular host language. This chapter includes brief examples of syntactic specifications, semantic specifications and the specifications that determine which modules are to be composed. It also describes how the various aspects of the higher-order attribute grammar formalism (described in Chapter 2) are mapped onto the features of the Silver specification language. This information will serve as background for the description in Chapter 4 of the ableJ system and process of writing composable semantic specifications for non-trivial languages.

Chapter Outline This chapter is outlined as follows:

- Section 3.1 explains the basic structure of Silver specifications, and describes Silver modules and Silver files.
- Section 3.2 describes the process of specifying concrete syntax in Silver's specification language, and the associated Copper tool, which is an integrated parser-scanner generator.
- Section 3.3 describes how the Silver specification language can be used to write

fragments of higher-order attribute grammars to define a language's abstract syntax and semantic analyses such as type-checking and identifier disambiguation.

- Finally, Section 3.4 describes how a host language Silver module can be composed with a specified set of extension modules to generate an executable compiler for a particular extended version of the host language.

More information about Silver is available at `http://melt.cs.umn.edu/silver/`. Readers familiar with Silver may skip ahead, and refer back to descriptions of individual features, as needed.

3.1 Silver Modules for Host Language and Extension Grammars

The Silver declarations for a host language's or extension's lexical and concrete syntax, as well as its abstract semantics, are grouped into Silver files (with the extension `.sv`) which are further organized into grammar modules.

3.1.1 Silver Modules

Figure 3.1 gives an overview of the various declarations in a Silver grammar module. These include the concrete syntax definitions described in Section 2.1.1 and the attribute grammar definitions shown in Figure 2.3. While this figure represents the grammar as a single file, in actual use, it would be split across multiple files in a directory. Each grammar module is named after a directory and defines a shared scope that includes all the declarations in the Silver files in that directory. Module names also incorporate Internet domains, in a manner similar to Java packages, to prevent name clashes. For example, the ableJ host language grammar is specified in the Silver module `edu:umn:cs:melt:ablej`, and its declarations consist of the Silver files in the directory `edu/umn/cs/melt/ablej/`. Modules group together related declarations such as the attribute grammar fragments that define the semantics of a host language, or the terminals, non-terminals and context-free productions that define the concrete syntax of the host language or of a new construct added by an extension. They thus define self-contained (extensible) specifications, either for a host language or an extension to

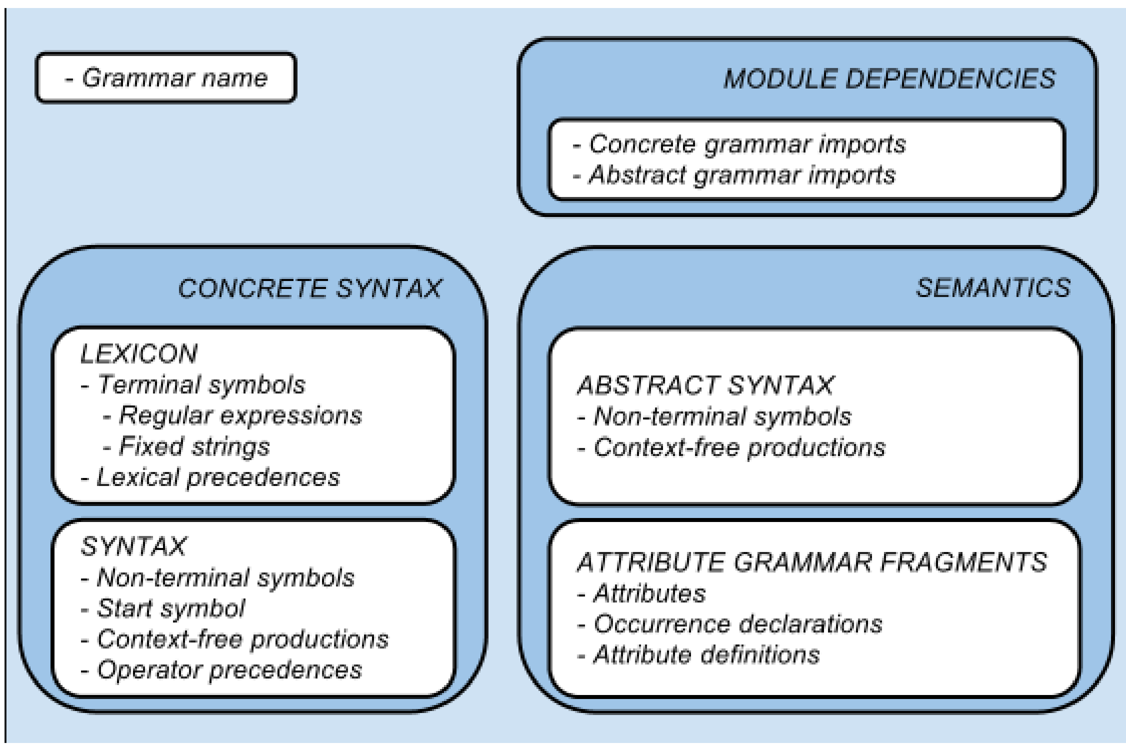


Figure 3.1: Overview of the declarations in a Silver grammar module.

an existing host language module.

3.1.2 Silver Files

As shown in Figure 3.2, each Silver file starts by declaring its grammar's name using the `grammar` keyword. This provides a means for other modules to import its declarations. This is followed by a list of `import` statements that allow the specification to reference definitions declared in other modules. The rest of the file consists of a list of declarations. Declarations may be single-line and delimited by semi-colons as in the case of terminals, or have bodies delimited with braces, as in the case of productions. Finally, double-slashes are used to write comments and indicate that all characters that follow up to the end of the line are to be ignored.

3.2 Extensible Concrete Syntax Using Copper

This section describes how Silver's specification language can be used to define the terminals, non-terminals and productions of a context-free grammar (see Section 2.1.1) that defines the syntax of a host language or extension construct. Figure 3.2 shows the Silver syntax specification of the language defined by the grammar in Figure 2.1. The Silver version of the grammar has fewer non-terminals and productions as it uses mechanisms to specify operator precedence and associativity.

We now look at each of Silver's concrete-syntax related constructs in the order in which they appear in Figure 3.2.

Non-Terminals The `nonterminal` keyword is used to specify the grammar's non-terminals. In Figure 3.2, `Expr` and `Defs` are declared to be non-terminals. `Expr` is flagged as the grammar start non-terminal using the `parser` construct.

Terminals Terminal symbols are defined using the `terminal` keyword in one of two ways. As shown in Figure 3.2, terminal symbols for fixed string keywords such as `let` and operator symbols such as `+` are declared by specifying the string in single quotes. Variable terminal symbols, on the other hand, are declared using standard regular expressions.

```
grammar g4;

nonterminal Expr;
nonterminal Defs;

parser parse :: Expr  g4;

terminal Let_t 'let' lexer classes { KEYWORDS };
terminal In_t 'in' precedence = 5, lexer classes { KEYWORDS };
terminal Plus_t '+' precedence = 10, association = left;
terminal Assign_t '=';
terminal Semi_t ';';

terminal Id_t /[a-zA-Z_\$][0-9a-zA-Z_\$]*/ submits to { KEYWORDS };
terminal IntConst_t /[0-9]+/;

ignore terminal WhiteSpace_t /[\t\ \n]+/;

lexer class KEYWORDS;

concrete production letExpr e::Expr ::= 'let' ds::Defs 'in' x::Expr { }
concrete production add e::Expr ::= l::Expr '+' r::Expr { }
concrete production idRef e::Expr ::= id::Id_t { }
concrete production intConst e::Expr ::= i::IntConst_t { }
concrete production consDefs d::Defs ::= id::Id_t '=' e::Expr';' ds::Defs{}
concrete production emptyDefs d::Defs ::= { }
```

Figure 3.2: Specifying concrete syntax in Silver.

For example, the terminal symbol for identifiers, `Id_t`, is declared to match all non-empty alpha-numeric sequences with an initial alphabetic character. Terminals may also specify lexical precedence, so that keywords for example, have lexical precedence over identifiers. This is needed so that the string `"let"`, which is matched by the regular expressions of both identifiers and the `Let_t` keyword, is assigned to the latter. Finally, the `ignore terminal` construct can be used to define comment-type tokens that will be ignored by the scanner and not sent to the parser.

Productions Concrete productions are declared using the `concrete production` construct. Each production declaration includes a production name, a production signature and a production body. Fixed string terminals can be represented directly by quoted strings in grammar signatures. Parameters in production signatures are assigned names using the `::` construct. For example, in Figure 3.2, the parameter `e` on the right-hand side of the signature of the production `consDefs` is declared to be of type `Expr`. The production body includes attribute definitions (elided in this figure and described in Section 3.3.1) that may reference such named signature parameters. Terminal symbol declarations may specify operator precedence and associativity information that can be used by the parser to handle ambiguities among grammar productions. This allows the grammar in Figure 3.2 to avoid the use of extra non-terminals such as `Term` and `Factor`.

Parser-Scanner Generation The concrete syntax declarations in each Silver file are used to generate the scanner and parser for the host language or for the host language extended with new constructs. Silver handles concrete syntax specifications using Copper, an integrated parser and context-aware scanner generator developed by August Schwerdfeger [13, 14, 15] (see Section 1.1.3.1). The parser and scanner generator uses only certain Silver declarations, viz. those of concrete productions and of those non-terminals and terminals that are present in the signature of at least one concrete production. Declarations that are not relevant to the parser and scanner are ignored during parser generation. These include abstract productions (described below), attribute definitions, and any non-terminals and terminals that appear only in abstract productions.

3.3 Specifying Language Semantics via Higher-Order Attribute Grammars

In this section, we look at how Silver's specification language can be used to specify the higher-order attribute grammars described in Chapter 2. For a given attribute grammar specification, Silver performs type-checking to ensure that all of its attribute grammar definitions are correctly typed, as described in Section 2.2.2. It then generates Java class files that implement an executable attribute evaluator for the grammar. This evaluator can be used to compile source code written in the language specified by the attribute grammar. The generated evaluator is demand-driven, unlike the formal model of evaluation described in Section 2.3 which assumes that all attribute instances on the parser-generated syntax tree are evaluated. That model therefore subsumes all the attribute evaluation sequences that are possible with the Silver evaluator.

3.3.1 Specifying Attribute Definitions on Grammar Productions

Silver's specification language includes constructs for declaring attributes and for specifying attribute definitions within grammar productions. Figure 3.3 shows the Silver version of the attribute grammar shown in Figure 2.4. The figure shows how attribute definitions are added to the Silver concrete syntax specification in Figure 3.2. As shown in the figure, attributes can be defined directly on the concrete productions described in the previous section. But in most large specifications (such as the ABLEJ grammar described in Chapter 4), we write attribute definitions on abstract productions.

Abstract Productions These productions, declared using the `abstract production` construct, are used only in the compiler's internal analyses and are not used in parser generation. In Figure 3.3, `consEnv` and `emptyEnv` are abstract productions that are used to construct symbol tables.

Synthesized and Inherited Attributes Figure 3.3 illustrates how attributes are declared in Silver's specification language. Each attribute declaration includes the kind of attribute (`synthesized` or `inherited`) and the attribute type. For example `value` is an `Integer`-valued synthesized attribute while `lookFor` is a `String`-valued inherited

```
synthesized attribute defs :: Env;
synthesized attribute value :: Integer;
synthesized attribute found :: Boolean;
inherited attribute env :: Env;
inherited attribute lookFor :: String;

nonterminal Expr with value, env;
nonterminal Defs with defs;
nonterminal Env with value, found, lookFor;

concrete production letExpr e::Expr ::= 'let' ds::Defs 'in' x::Expr {
  e.value = x.value;
  x.env = ds.defs;
}
concrete production add e::Expr ::= l::Expr '+' r::Expr {
  e.value = l.value + r.value;
  l.env = e.env;
  r.env = e.env;
}

concrete production idRef e::Expr ::= id::Id_t {
  local attribute ee :: Env = e.env;
  ee.lookFor = id.lexeme;
  e.value = if ee.found then ee.value
            else error ("Unknown identifier: " ++ id.lexeme);
}
concrete production intConst e::Expr ::= i::IntConst_t {
  e.value = toInt (i.lexeme);
}

concrete production consDefs d::Defs ::= id::Id_t '=' e::Expr';' ds::Defs {
  d.defs = bindingEnv (id, intConst (terminal (IntConst_t,
                              toString (e.value))), ds.defs);
}
concrete production emptyDefs d::Defs ::= { d.defs = emptyEnv (); }

abstract production bindingEnv e::Env ::= id::Id_t x::Expr rest::Env {
  e.found = id.lexeme == e.lookFor || rest.found;
  e.value = if id.lexeme == e.lookFor then x.value else rest.value;
  rest.lookFor = e.lookFor;
}
abstract production emptyEnv e::Env ::= { e.found = false; }
```

Figure 3.3: Writing attribute definitions to specify semantics in Silver.

attribute. These are both primitive typed attributes. `defs` and `env`, on the other hand, are higher-order attributes with non-terminal types, that evaluate to tree term values. Occurrence relations are specified within non-terminal declarations using the `with` construct. For example, the `defs` attribute occurs on the `Defs` non-terminal.

Local Attributes Local attributes are declared within productions using the `local attribute` construct. In Figure 3.3, the `idRef` production has a local `ee` of type `Env`.

Attribute Definitions and Expressions Silver's specification language includes syntax for specifying attribute definition expressions. Silver's syntax allows for writing attribute definitions that include all the various kinds of expressions shown in Figure 2.3, such as signature tree variables, attribute accesses, production and function calls, terminal symbols and conditional expressions. The Silver specification language includes a set of primitive types such as `String`, `Integer` and `Boolean`, primitive values such as `true` and `-1`, and primitive functions such as `toInt` and `+`.

3.3.2 Additional Constructs for Modular Composable Specifications

Silver's attribute grammar specification language includes several non-standard convenience features. In Chapter 4, we will show the usefulness of these features in the writing of modular extension specifications.

Forwarding Forwarding expressions in productions [23] are optional higher-order (i.e., tree-valued) expressions that define special tree values whose root non-terminals are the same as the productions' left-hand sides. They are declared using the `forwards to` construct. During attribute evaluation, the value of any synthesized attribute instance for which there is no explicit definition on the forwarding production, is evaluated on the root of the forwarding expression's evaluated syntax tree. Forwarding is a means to add implicit definitions for some attribute occurrences on a production while providing explicit definitions for all other attribute occurrences.

Aspect Productions These productions, declared using the `aspect production` construct, are special productions that allow extension writers to add attribute definitions to existing abstract or concrete productions. Extensions can therefore add new functionality (such as a new translation) to existing constructs by declaring new attributes and using aspect productions to add their definitions to existing productions. This makes it easier to write modular extension specifications.

Collection Attributes Collection attributes (inspired by Boyland [24]) are local attributes whose values are computed by combining the evaluated values of multiple definitions. A collection attribute is declared in a non-aspect production with a collection operator, using the `collect with` construct. Any aspect to that production may define a contribution to the collection attribute's value. The initial value of the collection attribute is specified in the (non-aspect) production in which it is declared, using a special `:=` assignment operator. Extensions can contribute elements to the attribute's value via aspect productions using a special `<-` assignment operator. The final evaluated value of the attribute is computed by combining all these individual values using the collection operator. The order in which the individual values are combined is undefined.

Pattern-Matching on Tree Terms Silver also allows for pattern-matching on tree values using production names [25]. Non-terminals assume the role of algebraic datatypes. Their value constructors are the various productions with the non-terminals on their left-hand sides. Attribute definitions can pattern-match on tree values using the `case` construct.

Functions Finally, functions, declared using the `function` keyword, are similar to productions except that the left-hand side symbol in the signature is replaced with a return type, and the body includes a `return` definition. Functions can be used to add elements to and retrieve elements from the program environment, as shown in Figures 4.8 and 4.10.

Polymorphism Silver supports parametric polymorphism [26]. It allows for declaring polymorphic non-terminals and attributes with type variables. It also includes special syntax for type-safe polymorphic lists.

Reference Attributes Reference attributes [27] evaluate to pointers to the roots of fully attributed trees. They can therefore be used to access the values of synthesized attribute instances on previously attributed trees. In Silver, reference attribute values are referred to as `Decorated` trees. While they are implemented as references or pointers, their usage is semantically equivalent to passing decorated trees between different nodes.

3.4 Composing Host Language Specifications and Extension Specifications

In Figure 1.1 we examined a hypothetical situation where a programmer uses an extensible language tool such as Silver to generate a compiler for an extended version of Java 1.4 augmented with two features: an enhanced `for` loop and a complex number type. Here we look at how this composition process is specified in Silver. ABLEJ, an extensible specification of Java 1.4, is described in Chapter 4.

The module `edu:umn:cs:melt:ablej` specifies the ABLEJ host language grammar, the module `edu:umn:cs:melt:ablej:extensions:foreach` defines the extended `for` loop extension and the module `edu:umn:cs:melt:ablej:extensions:complex` defines the complex number type extension. Figure 3.4 shows a specification that defines a version of the Java 1.4 host language extended with these two extensions. The components of a specific extended version of a language are specified by importing the appropriate modules into a single Silver file. The extended grammar in this example is specified by a driver file in the module `edu:umn:cs:melt:composed:ablej_foreach_complex`.

Host language and extension declarations are imported by naming the appropriate modules with the `imports` and `exports` constructs. The concrete syntax of the extended language is defined by specifying the host language and appropriate extension modules, and the grammar start symbol, within a `parser` declaration. The imported concrete syntax declarations are used in generating the scanner and parser for the extended language. If a module is imported but not included in the `parser` declaration, its nonterminals and productions are used only to generate the extended language's attribute evaluator. They are not incorporated into its parser. The driver file defines a special `main` function that calls the appropriate parser on input files using the `parse` keyword.

The Silver compiler reads in the Silver files in all referenced modules, verifies that

```
grammar edu:umn:cs:melt:composed:ablej_foreach_complex;

exports edu:umn:cs:melt:ablej;
exports edu:umn:cs:melt:ablej:extensions:foreach;
exports edu:umn:cs:melt:ablej:extensions:complex;
imports edu:umn:cs:melt:ablej only Root;

parser extparser :: Root {
     edu:umn:cs:melt:ablej;
     edu:umn:cs:melt:ablej:extensions:foreach;
     edu:umn:cs:melt:ablej:extensions:complex;
}

function main
IOVal<Integer> ::= args::[ String ] ioIn::IO {
  local attribute result::ParseResult<Expr> =
             parse (implode (" ", args), "");
  local attribute root::Root = result.parseTree;

  return ioval (if result.parseSuccess then print ("pp: " ++ root.pp, ioin)
             else print ("Parse error: " ++ result.parseErrors, ioin), 0);
}
```

Figure 3.4: Composing the Java 1.4 host language Silver specification with the enhanced `for` and complex extension specifications.

all required modules are available, and checks for any Silver syntax or type errors. It determines if a valid parser and attribute-evaluator can be generated, and if there is a valid `main` function. It then generates Java class files that implement the integrated parser-scanner and the attribute evaluator for the extended language.

Chapter 4

Writing Composable Specifications for Extension Semantics

4.1 Introduction

In this chapter, we describe the first contribution of the dissertation. We show how to design a host language keeping in mind the need for easy extensibility and safe composability to increase the likelihood that generated compilers act as expected. First, in Section 4.1.1, we give an overview of the dissertation's goal of composable semantics via higher-order attribute grammars. In Section 4.1.2, we look at examples of language extensions to get a better idea of the problem requirements and to derive a more specific problem statement. In Section 4.1.3, we define the problem as designing an extensible host language in a higher-order attribute grammar framework such as Silver, to allow for useful and composable extensions that add new constructs and analyses while interacting closely with the host language type and environment. We also list criteria for success for any candidate solution. In Section 4.1.4, we give an overview of our solution with a description of ableJ, an extensible specification of Java 1.4. Section 4.1.5 gives an outline of the rest of the chapter, which provides details about ableJ's design, including the use of forwarding and aspect productions to write modular extension semantic specifications, and about the design features of the host language's type and environment that allow extension writers to conveniently specify the semantics of new constructs.

4.1.1 Context and Goal: Composable Semantics via HOAGs

In Chapter 1, we discussed the usefulness of a declarative framework for incremental extension of languages (similar to libraries). Further, our goal of extension composability implies we are interested in composable syntax and semantics specifications that can be used to automatically construct sound and terminating compilers for extended versions of languages. This dissertation focuses on the problem of writing composable semantics. The syntactic aspects have been dealt with elsewhere [13, 14, 15].

In Chapter 3, we described Silver: an extensible language tool in which language semantics are specified using the higher-order attribute grammar formalism described in Chapter 2. Higher-order attribute grammars extended with features such as forwarding, aspect productions and collection attributes are an appropriate tool for dealing with the expression problem. New constructs can be added via new productions and new functionality to existing languages can be added via new attributes and aspect productions. Thus Silver provides a foundation for handling the expression problem. With simple macro-type extensions (such as adding a Java 1.5 style enhanced `for` loop to a Java 1.4 language specification), this is sufficient for building composable extensions. Not much is required of the host language design as far as providing hooks for potential extensions.

But as extension complexity increases, it is not clear how extensions may access and modify the host language's semantic analyses in ways that do not clash with other extensions doing the same. Even with a framework such as Silver, there are questions about how basic aspects of programming language semantics (such as types, environments and error-reporting) would be designed for a non-trivial language to allow extension writers to easily write extensions that compose with other independently written extensions. Further, it is desirable to do this while re-using host language semantics, to increase modularity and composability. These issues are explored in this chapter in the context of designing an extensible specification for the Java 1.4 programming language [28].

4.1.2 Three Examples of Extensions

We first look at a few examples of extensions in order to get a sense of the features required of an extensible host language specification. The requirements are driven primarily by how one might write equivalent Java 1.4 versions of the code. This lets us

```
Member[] group;
...
for (Member m:group) {
     m.process ();
}
```

```
Member[] group;
...
for (int i = 0; i < group.length; i++) {
     Member m = group[i];
     m.process ();
}
```

```
ArrayList group;
...
for (Member m:group) {
     m.process ();
}
```

```
ArrayList group;
...
for (Iterator it0 = group.iterator ();
        it0.hasNext ();) {
     Member m = (Member) it0.next ();
     m.process ();
}
```

Figure 4.1: Code that uses the enhanced `for` construct to iterate over arrays and collections, and equivalent Java 1.4 code.

formulate requirements on the design of the host language's semantics.

4.1.2.1 Adding an Enhanced `for` Loop

Our first extension augments Java 1.4 with a construct identical to the enhanced `for` loop introduced in Java 1.5 [29]. This is an example of a simple and easy-to-understand extension that a programmer might want to add to Java without waiting for a major update of the compiler. The enhanced `for` loop allows for iteration in source code over arrays, and over collections of discrete elements such as lists. The left column of Figure 4.1 shows examples of code using this construct to perform both kinds of iteration.

Equivalent Java 1.4 versions of the code can be written using the standard `for` loop (as shown in the right column). While Java 1.5 provides the iteration facility on objects of type `Iterable`, the equivalent Java 1.4 code shown here uses the `Collection` interface and its `iterator`, `next` and `hasNext` methods to loop over group elements. Iteration over arrays using the enhanced `for` loop can be implemented in Java 1.4 by looping over the arrays using numerical indices.

This extension illustrates the minimum requirements of any extensible language framework.

- A basic requirement is a way to specify new syntax and terminals. In this dissertation, we do not focus on the aspects of language extensibility that deal with concrete syntax. Extension writers must also be able to extend the host language's abstract syntax with new abstract productions for any new constructs.
- A second requirement is a way to easily construct valid tree values, to compute the host language translation.
- The extended language's compiler must statically check each loop expression's type and verify that it is an iterable value. The extension must therefore be able to access the type information that is computed by the host language specification. Further, it must be able to generate appropriate equivalent host language code based on this type.
- The extension must also be able to add and retrieve elements from the environment, so that the loop variable can be added to the loop body's environment.
- Finally, it must be possible for the extension to use the results of any type-checking or symbol look-ups to provide the programmer with error messages that are appropriate for the construct, and not for the generated Java code.

4.1.2.2 Adding Complex Numbers as a Primitive Type

This extension adds complex numbers as a primitive type to Java 1.4. The first code fragment in Figure 4.2 is written in a version of Java extended with this type. The extension provides a new construct to declare complex-typed variables, using the `complex` keyword. Complex literals are specified using the syntax

`complex (` *real-part*`,` *imag-part* `)`

The extension performs overloading of operators such as addition, and automatic coercion of values from existing types (such as double) to the new type. Equivalent Java 1.4 code would use the `Complex` class, as shown in the second code fragment in Figure 4.2. This extension's requirements of an extensible language framework are as follows.

- The extension must be able to add constructs for new types, as well as new kinds of expressions, such as complex literals.

```
complex c, d;
c = complex (1,-2);
d = c + 2.7;
System.out.println (c + d);
```

```
Complex c, d;
c = new Complex (1,-2);
d = new Complex (c.real() + 2.7, c.imag());
System.out.println(new Complex(c.real() + d.real(), c.imag() + d.imag()));
```

Figure 4.2: Code in which complex numbers are a primitive Java type, and equivalent Java 1.4 code.

- The extension must be able to statically check that the real and imaginary components of complex literals are valid double expressions.
- Variable declaration information of the new type must behave like existing environment items in how they are added to and retrieved from symbol tables.
- The extension must be able to plug the new type into the existing type-hierarchy, by specifying new sub-typing relationships (say from double values to complex values), and by performing run-time coercion from existing types to the new type, say by adding a zero-valued imaginary component to double values.
- The extension must be able to overload existing operators.

While these requirements deal with how extension writers might specify new semantics, they also raise the question of the composability of extension semantic specifications. For example, what assurances can be given that the addition or retrieval of environment items by one extension will not conflict with those by another extension. What guarantees can be given that the addition of new sub-typing relations to the host language type hierarchy will not result in circularities? While we do not aim to develop static analyses to solve all these challenges, it would be highly desirable to come up with best practices for extension writers to follow, that would ensure that their extensions work properly with other well-behaved extensions.

4.1.2.3 Adding Auto-Boxing and Auto-Unboxing

```
int i = 0;
Integer J = i;
int k = J;
```

```
int i = 0;
Integer J = new Integer(i);
int k = J.intValue();
```

Figure 4.3: Code that uses auto-boxing and auto-unboxing, and equivalent Java 1.4 code.

This extension adds automatic boxing and unboxing of primitive types to Java 1.4. This allows for automatic conversion of primitive values to and from their corresponding object class values, in contexts such as assignments and method calls. It helps in writing more concise code. The first code fragment in Figure 4.3 uses auto-boxing. The second fragment shows equivalent Java 1.4 code in which conversion is explicit. Since conversion occurs between primitive types and object types in both directions, this is a situation where a coerced value's type may not be a sub-type of the target type. Thus the host language type hierarchy must be designed for new relations to be added via mechanisms other than sub-typing.

4.1.3 Problem: Composable Semantics via Higher-Order AGs

The problem explored in the first part of this dissertation is that of designing an extensible host language in a higher-order attribute grammar framework such as Silver, in a way that allows for extensions more interesting than simple macro-style additions.

Extensions may specify domain-specific analyses for the constructs they add or new analyses for existing features. They may:

- add new constructs such as statements, types and expressions,

- access information from the host language type system, plug in new types into the host language type-hierarchy, specify new sub-typing relationships and new run-time type conversion mechanisms, perform coercion in contexts such as assignment, and specify operator overloading,
- add and retrieve elements from the environment, and add new kinds of declaration information to the program environment in a safe and composable manner, and
- construct valid syntax tree values to specify host language translations or the results of source-code transformations, such as optimizations based on static compile-time analyses.

4.1.3.1 Criteria for Success

The notion of a "well-designed extensible language specification" is necessarily informal. It depends on the constantly changing needs of programmers and the domains of the problems they must solve. We cannot therefore be completely formal or rigorous when setting down the criteria for success by which we will judge any candidate extensible language framework. But we can list the following guidelines:

1. The framework must allow for host language specifications that are easily extensible by extension writers with limited experience with the framework.
2. Extension writers must be able to write semantic specifications in a modular fashion. They must be able to re-use host language semantics, while at the same time specifying additional extension semantics.
3. Strong evidence for the usefulness of any such framework would be a numerous set of useful, diverse and interesting extensions.
4. Programmers must be able to use the framework to generate compilers for desired versions of host languages, without having expertise either in the extension domains, or in the general areas of programming languages and compilers. Composition should be as close as possible to being automatic.
5. New constructs must have the same look-and-feel as constructs in the host language and fit comfortably within the host language syntax.
6. The extended language's compiler should generate errors in terms of the higher-level constructs used by the programmer, and not in terms of the generated host

language translation (as with macros). This would shield programmers from having to analyse generated source code.

7. Given our goal of composability, independently written extensions must compose.
8. Extensions must be statically analysable, preferably modularly, to guarantee termination of the generated compilers.

4.1.4 Solution: ableJ: An Implementation of Composable Semantics

In this chapter, we present an approach that satisfies the above criteria and deals with one facet of the expression problem, that of writing composable semantic specifications for language extensions. We show that a declarative, higher-order attribute grammar-based tool, such as Silver, extended with features such as forwarding, aspect productions and collection attributes, can be used to write useful, modular and composable extension semantic specifications.

ableJ [19] is an extensible language framework for Java 1.4 specified using the Silver tool. It can be used to generate front-end translators from code written in Java extended with various features, to code in pure Java 1.4. The ableJ host language specification is designed for extensions that interact closely with the host language's type system and environment, and return error messages that reference extended code, rather than generated Java code. Extensions written to ableJ range from a simple macro-like enhanced `for` extension, to more interesting ones for complex number types, auto-boxing and algebraic data-types with pattern-matching. These independently written extensions can be combined when the extended language's compiler is constructed. As described in Chapter 5, most of them are analysable for termination of higher-order attribute evaluation. Some also pass the modular version of the analysis.

Our experience designing and implementing ableJ has given us insight into the planning that is essential to writing a host language specification that can be conveniently extended with useful extensions. We can also draw conclusions about the kinds of extensions that are appropriate to frameworks such as Silver. While we do not aim to formally show soundness of the generated compiler, or its termination in the general sense, we provide the output of generated compilers to show that the specifications are correct, and that the extensions work together as expected. In Chapter 5, we will consider the problem of formally termination of the process of higher-order attribute evaluation during the execution of the generated compiler.

4.1.5 Chapter Outline

This chapter is outlined as follows:

- Section 4.2 gives an overview of the ableJ framework.
- Section 4.3 looks at how the various features of the Silver tool can be used to handle the expression problem, by showing how new constructs and analyses can be added to Java 1.4.
- Section 4.4 looks at how the host language's environment is designed to allow extensions to access program symbol tables, and add their own environment items without clashing with other extensions.
- Section 4.5 looks at the design of the host language's type system and demonstrates how extensions may add new types and sub-typing relations to the existing Java 1.4 type hierarchy, specify mechanisms for run-time conversion, and perform operator overloading.
- Section 4.6 describes a few additional extensions that we have written to ableJ.
- Section 4.7 concludes with a discussion of the ableJ framework.

4.2 ableJ: An Extensible Specification of Java 1.4

Specified using the Silver tool (described in Chapter 3), ableJ [19] is an extensible language framework for Java 1.4 [28] in which language semantics are specified using the higher-order attribute grammar formalism [17]. The ableJ host language specification consists of a set of Silver modules that

- define Java 1.4's concrete syntax,
- define Java 1.4's abstract syntax using higher-order attribute grammars, and
- define a "driver" function that is executed when the generated compiler is run on source code.

When generating a compiler for a version of Java 1.4 extended with multiple extensions, the driver module specifies the list of desired extensions that are to be composed with the host language. Extensions may define new concrete syntax, as well as new abstract syntax and semantic analyses using higher-order attribute grammars. The Silver compiler composes the host language and all specified extension modules, and generates an executable compiler for the extended language. This compiler includes an integrated parser-scanner, and an attribute evaluator that performs semantic analysis and translation to pure Java 1.4.

In ableJ, semantic analysis is done solely on host language and extended language source code. Host Language and extension specifications do not define or manipulate byte-code. The generated compiler is a pre-processor or source-level transformer that takes code written in an extended version of the host language, performs type-checking and other semantic analyses on it, and then generates a valid Java 1.4 translation. If an extension adds a new construct, it is expected to also define its translation to equivalent code in Java 1.4. This practice aids in the modularity of the framework and is enforced in Silver by a modular attribute grammar well-definedness analysis [18]. The translated Java 1.4 code can be compiled using a traditional Java compiler to executable byte-code. Extensions may perform their own compile-time analyses to ensure that the generated host language translation is free of type errors, access violations and other errors.

In this chapter, we consider those aspects of the specification that allow for interesting, modular and composable extension specifications. We focus in particular on the use of forwarding for modular composable semantics, on the design of the program environment, and on the extensible type system. We show only a few snippets of the ableJ grammar in this chapter. It is described in more detail in [19]. The full specification is available for download at `melt.cs.umn.edu/software.html`.

4.3 Modular Semantics to Handle the Expression Problem

In this section, we demonstrate how higher-order attribute grammar frameworks extended with features such as forwarding, aspect productions and collection attributes provide a foundation to solve the expression problem. We first show how they may be used to extend host languages with simple, macro-style extensions. As extensions get more complex, these features can still be used, but only in conjunction with restrictions on extension specifications. When more interaction with the host language's type and environment is needed, other features such as pattern-matching may be used, as described in the sections that follow.

4.3.1 New Constructs via Higher-Order Attributes with Forwarding

Forwarding [23] is an addition to higher-order attribute grammars that aids in writing modular extension specifications. Forwarding expressions in productions are optional higher-order (i.e., tree-valued) expressions that define special tree values whose root nonterminals are the same as the productions' left-hand sides. During attribute evaluation, the value of any synthesized attribute instance for which there is no explicit definition on the forwarding production, is evaluated on the root of the forwarding expression's evaluated syntax tree. Forwarding is a means to add implicit definitions for some attributes on a production while providing explicit definitions for all other attributes. It allows extensions to specify translations (specified as higher-order tree values) to equivalent host language code and re-use host language semantics, which aids in writing modular language specifications, while at the same time allowing for explicit attribute definitions that specify additional domain-specific analyses.

For example, a query in a version of Java embedded with SQL (described in Section 4.6.2) could be translated to equivalent host language code that calls a method in the JDBC library method. Thus in the SQL extension specification, the query construct's production forwards to a valid syntax tree value for the JDBC method call. The query can perform additional specialized static error checking, while obtaining all other analysis results from the equivalent host language tree.

Another extension that uses forwarding is one (described in Section 4.1.2.1) that extends Java 1.4 with an enhanced `for` loop, similar to that introduced by Java 1.5. Figure 4.4 shows a fragment of this extension's specification. The `for_each` production adds the new construct's syntax to the existing concrete grammar. Some existing attributes are defined explicitly on this production, while others are defined implicitly using forwarding. The production defines the `pp` and `errors` attributes explicitly, so as to provide useful programmer feedback. The forwarded to tree is constructed based on whether the iteration is over an array or over a collection, or whether instead there is a type error. Code for iteration over arrays is defined by the local attribute `forOverArray` which increments an integer array index after each loop iteration. This generated code

```
grammar edu:umn:cs:melt:ablej:extensions:foreach;
import edu:umn:cs:melt:ablej;

concrete production for_each
foreach::Stmt::= 'for' '('type::Type var::Id':' group::Expr')' body::Stmt {

   foreach.pp = "for (" ++ type.pp ++ " " ++ var.lexeme ++ ":" ++
                  group.pp ++ ")\n" ++ body.pp;

   foreach.errors = if isCollection || isArray then [ ]
         else [ "for loop must iterate over Collections or arrays." ];

   forwards to if isCollection then forWithIterators
                  else if isArray then forOverArray
                  else skip ();

   local attribute isCollection :: Boolean;
   isCollection = ...; - Check if var's type is a sub-type of Collection.

   local attribute isArray :: Boolean;
   isArray = ...; - Check if var is an array.

   local attribute forWithIterators :: Stmt;
   forWithIterators = ...; - Iterate using Collection methods.

   local attribute forOverArray :: Stmt;
   forOverArray = ...; - Iterate using array index.
}
```

Figure 4.4: A Silver specification that adds an enhanced `for` loop to Java 1.4, and uses forwarding to translate extended code to pure Java 1.4 code of the form shown in Figure 4.1.

```
grammar edu:umn:cs:melt:ablej:extensions:complex;
import edu:umn:cs:melt:ablej;

terminal Complex_t 'complex';

concrete production complex_type
type::Type ::= 'complex' {
     type.pp = "complex";
     type.errors = [ ];
     forwards to ...; - The type of the Complex class.
}

concrete production complex_literal
literal::Expr ::= 'complex' '(' real::Expr ',' imag::Expr ')' {
     literal.pp = "complex (" ++ real.pp ++ ", " ++ imag.pp ++ ")";
     literal.errors = ...; - Verify that real and imag are double.
     forwards to new_object("Complex", cons_expr_list (real,
           cons_expr_list (imag, empty_expr_list ())));
           - "new Complex (real, imag)"
}
```

Figure 4.5: A Silver specification that uses forwarding and adds complex numbers as a primitive type to Java 1.4.

is similar to that in the top right of Figure 4.1. Code for iteration over collections is defined by the local attribute `forWithInterators` which calls the `Collection` interface's methods. This generated code is similar to that in the bottom right of Figure 4.1. If the iterated value is neither a sub-type of `Collection` nor an array, an error message is generated and the for-each loop forwards to a `skip` statement.

The complex number extension also uses forwarding in its specification of new types and literals, as shown in Figure 4.5. The `complex_type` production forwards to references to the Java class `Complex`. Similarly, the `complex_literal` production forwards to a call to a constructor for the `Complex` class after verifying that the literal's real and imaginary components are valid double expressions.

Finally, we look at how forwarding can be used to construct the syntax tree corresponding to a program's host language translation. This could be used to perform a

```
grammar edu:umn:cs:melt:ablej;
start nonterminal Root;
nonterminal Expr, Stmt, Type;

synthesized attribute pp :: String occurs on Expr, Stmt, Type;
synthesized attribute errors::[String] collect with ++ occurs on Expr,Stmt;
synthesized attribute hostStmt :: Stmt occurs on Stmt;
synthesized attribute hostExpr :: Expr occurs on Expr;
synthesized attribute hostType :: Type occurs on Type;

concrete production while
loop::Stmt ::= 'while' '(' cond::Expr ')' body::Stmt {
     loop.pp = "while (" ++ cond.pp ++ ") \n" ++ body.pp;
     loop.errors := ...; - Check that the condition is boolean.
     loop.hostStmt = while ('while', '(',cond.hostExpr,')', body.hostStmt);
}

concrete production local_var_dcl
dcl::Stmt ::= type::Type id::Id_t ';' {
     dcl.pp = type.pp ++ " " ++ id.lexeme ++ ";";
     dcl.errors := ...; - Check that id is not declared elsewhere.
     dcl.hostStmt = local_var_dcl (type.hostType, id, ';');
}

abstract production skip
stmt::Stmt ::= {
     stmt.pp = ""; stmt.errors := [ ]; stmt.hostStmt = skip ();
}

concrete production reference
expr::Expr ::= id::Id_t {
     expr.pp = id.lexeme;
     expr.errors := ...; - Check that id has a valid declaration.
     expr.hostExpr = reference (id);
}

concrete production double_type
type::Type ::= 'double' {
     type.pp = "double";
     type.hostType = double_type ('double');
}
```

Figure 4.6: ableJ definitions that construct the syntax tree term corresponding to an extended Java program's host language (pure Java 1.4) translation.

sanity check for errors on generated code. Each non-terminal X is decorated by a synthesized attribute `host`X of type X. Figure 4.6 shows the definitions of these attributes on a few representative Java 1.4 host productions. For conciseness, the figure shows the definitions on concrete productions; in the actual specification, these are defined on the abstract versions of these productions. Each `host`X attribute instance stores the host translation corresponding to its sub-tree. Thus on the root production, `hostRoot` evaluates to the Java 1.4 equivalent of the entire program. These attributes are defined on host productions, but not on extension productions. The value of these attributes within extension constructs are retrieved off their forwarded-to nodes, which generally define equivalent sub-trees in the host. The definition of these attributes, in conjunction with the semantics of forwarding, propagates the translation process at an extension construct's tree node down to its child sub-trees. Thus querying `hostRoot` off the root of a tree constructed in an extended version of Java 1.4, returns the host equivalent of the entire program.

4.3.2 New Functionality via Aspects and Collection Attributes

Aspect productions (described in Section 3.3.2) are a mechanism by which new attribute definitions can be added to existing productions. Aspect productions are ideal for adding new functionality to existing constructs, say to define a new translation for an existing language. In Silver new constructs can be added via the specification of new concrete and abstract productions which can be composed with the original grammar specification. In combination with the ability to add new constructs, aspect productions can thus be used to handle one facet of the expression problem. More interesting than the problem of adding new functionality to existing constructs, is that of extending functionality that already exists in the host language, for then we must deal with questions regarding potential interaction or conflicts between different extensions.

There are situations where the likelihood of improper interaction between extensions is low. An example is error message generation, in which the order of messages generated by distinct extensions is not relevant. Any interaction between lists of messages generated by constructs in different extensions is limited to list concatenation. In such simple situations, extensions can extend existing host language functionality via aspect productions and collection attributes. For example, the ableJ host language specification includes a collection attribute `errors` whose collection operator is the list concatenation function. The host language grammar specifies initial values of the `errors` attribute on its productions. These are based on the results of the type-checking and binding analysis performed on the nodes of the program syntax tree. Extensions can add their own analysis results and error messages. Figure 4.4 shows an example in which the enhanced `for` extension adds an error message if the loop expression is not an iterable value. These individual error messages are collected together and returned to the programmer as a single list of errors. Thus aspect productions and collection attributes are an appropriate mechanism for extending existing functionality in simple situations such as having extensions specify additional error messages.

A more interesting example of extending existing functionality is the addition of new sub-typing relations and associated conversion mechanisms. In any extended version of the host language, the sub-typing relation must be reflexive, anti-symmetric and transitive. In other words, there may be no circularities within the type hierarchy. This property may not hold, even when we compose extensions that do not individually introduce circularities into the host language's type hierarchy. For example, consider the situation described below, where $t_1 \prec t_2$ indicates that the type t_1 is a sub-type of the type t_2:

- the host language specification defines two unrelated types h_1 and h_2,
- an extension specification E_1 defines a new type e_1 and adds the relations $h_1 \prec e_1$ and $e_1 \prec h_2$, and
- an extension specification E_2 defines a new type e_2 and adds the relations $h_2 \prec e_2$ and $e_2 \prec h_1$.

Each extension independently composes with the host language without introducing any circularities into the type hierarchy, but when the host language is composed with both extensions simultaneously, a circularity is introduced.

This problem would be avoided if every extension adhered to the condition that no extension type can be both a sub-type of a host language type, and a super-type of a host language type. This is an invariant specified informally by the host designer, that if satisfied by extension writers, guarantees that their extensions compose with other well-behaved extensions. Section 4.5.2 gives more details about sub-typing in ableJ.

While these invariants are not statically checked by the Silver tool, they are similar to the modularity condition on extensions described in Chapter 5 that ensures that an extension passes the monolithic higher-order attribute evaluation termination analysis when composed with other extensions that similarly satisfy it.

Thus aspect productions and collection attributes can be used to extend existing host language functionality in interesting ways, but only in conjunction with restrictions on extension specifications.

4.4 Designing the Host Language Environment for Composable Extensions

Semantic analysis requires information flow between different parts of the syntax tree. For example, information derived from the declarations of fields, identifiers, methods and classes about types and accessibility must be available during expression type-checking. In attribute grammar frameworks, this information is passed between the nodes of the syntax tree using both synthesized and inherited attributes. ableJ provides a set of attributes, non-terminals, productions and functions that encapsulate semantic information at the point of declaration into scoped lists of environment items, pass them around the syntax tree and retrieve information where needed.

Extensions may add their own environment items at the appropriate scope. If they define new types, they may also define new kinds of binding information. For example, the SQL extension (described in Section 4.6.2) defines table columns in its `import table` construct that are accessible within its `using ... query` construct. The environment must be defined correctly when the ableJ host language is composed with multiple extensions. For example, consider the situation in which the ableJ host language is composed with both the enhanced `for` and SQL extensions. An SQL query inside an enhanced `for` loop must be able to properly retrieve the column definitions added by a preceding `import table` construct from the program environment. The host language environment is therefore designed to allow extensions to add their own environment items at the appropriate scope in a composable fashion.

4.4.1 Implementing the Host Language Environment

```
grammar edu:umn:cs:melt:ablej;

- Environment items are the basic unit of binding information.
nonterminal EnvItem;
abstract production var_binding
envItem::EnvItem ::= name::String rep::TypeRep { ... }

- The list of environment items generated from declarations.
synthesized attribute defs :: [ EnvItem ];

- Constructing environment items at the point of declaration.
concrete production local_var_dcl
dcl::Stmt ::= type::Type id::Id_t ';' {
     dcl.defs = [ var_binding (id.lexeme, type.typeRep) ];
}
```

Figure 4.7: ableJ definitions that construct and collect environment items at the point of declaration.

Figure 4.7 shows some of the host language's declarations that can be used to create and access symbol tables. The basic units of binding information in ableJ are environment items of type `EnvItem`. They bind identifier names to type representations, among other values. These are constructed at the point of declaration from the identifier's lexeme and any declaration-specific information. For example, the local variable declaration production `local_var_dcl` in Figure 4.7 uses the `var_binding` production to create a simple environment item from its variable's lexeme and its declared type's `typerep`. The list of environment items generated within a node's sub-tree is gathered up by the synthesized attribute `defs`, as shown in the `local_var_dcl` production.

Environment items are passed down via the inherited attribute `env` (declared as shown in Figure 4.8). At each point in the program, this attribute specifies the set of definitions that are in scope at that point. In Java, environment items have to be added at the appropriate scope. Separate symbol tables are defined for top level declarations in a file, for declarations in other files in the package, for explicit single type imports, for on-demand imports, and for definitions within methods and inner classes. Environment

```
grammar edu:umn:cs:melt:ablej;

- The environment is a list of scopes, each with its own list of items.
inherited attribute env :: [ Scope ];
nonterminal Scope with scopeType, envItems;
synthesized attribute envItems :: [ EnvItem ];

- A function to append a new local scope to an existing environment.
function addLocalScope
[ Scope ] ::= items::[ EnvItem ] enclosingEnv::[ Scope ] { ... }

- The while loop passes the environment unchanged down the syntax tree.
concrete production while
loop::Stmt ::= 'while' '(' cond::Expr ')' body::Stmt {
     cond.env = loop.env;
     body.env = loop.env;
}
```

Figure 4.8: ABLEJ definitions that construct and pass scoped symbol tables down the syntax tree.

items are collected into scoped symbol tables at those nodes that define new scopes. On nodes that do not define new scopes, the environment is passed unchanged down the syntax tree, as in the `while` loop. An example of a function that creates a new scope is the `addLocalScope` function which creates a new local scope on top of a pre-existing environment. This is used by the enhanced `for` loop to append a new local scope (containing the loop variable) to the existing environment before it is passed down to the loop body, as shown in Figure 4.9. While the specification shown here implements the environment using lists, it could be implemented using conventional search trees for faster access.

Finally, the ABLEJ specification includes a set of functions (such as `lookUpVariable`) that retrieve information from the environment based on identifier lexeme. Figure 4.10 shows how the `reference` production uses the `lookupVariable` function to retrieve its identifier's binding information from its environment. An error message is generated if the identifier cannot be resolved to a single valid declaration.

```
grammar edu:umn:cs:melt:ablej:extensions:foreach;
import edu:umn:cs:melt:ablej;

- The loop body's environment adds a new scope with the loop variable.
concrete production for_each
foreach::Stmt::= 'for' '('type::Type var::Id':' group::Expr')' body::Stmt{
     type.env = foreach.env;
     group.env = foreach.env;
     body.env = addLocalScope ([ var_binding (var.lexeme, type.typeRep) ],
                               foreach.env);
}
```

Figure 4.9: An example of an extension (the enhanced `for` loop) that uses the ABLEJ definitions in Figure 4.8 to add binding information to the program symbol table.

```
grammar edu:umn:cs:melt:ablej;

- Retrieving an identifier's information from the environment by its name.
function lookUpVariable [ TypeRep ] ::= name::String env::[ Scope ] { ... }

- Retrieving information from the environment at a variable reference.
concrete production reference
expr::Expr ::= id::Id_t {
     local attribute lookupResults :: [ TypeRep ];
     lookupResults = lookupVariable (id.lexeme, expr.env);

     expr.typeRep = head (lookupResults);
     expr.errors = if (length (lookupResults) == 1) then [ ]
          else if (length (lookupResults) == 0) then [ "Undeclared id." ]
          else [ "Multiply declared id." ];
}
```

Figure 4.10: ABLEJ definitions for variable look-up in the environment and an example of their use.

4.4.2 Retrieving Environment Items Using Pattern Matching

As shown in Figure 4.9, the enhanced `for` extension adds environment items using existing definitions in the host language. However, extensions may define their own productions to construct environment items. Items added by a construct in one extension must be properly accessible by other constructs in that extension, even if they are passed between them via a construct in another extension. For example, an SQL query inside an enhanced `for` loop must be able to properly extract the column definitions added by a preceding `import table` construct.

We have implemented an extension to the Silver specification language that performs pattern-matching on tree values using production names. This allows extensions to access the program environment in a safe and composable manner. Suppose the SQL extension defines a new production to construct its environment items. The extension would use pattern-matching on this production's name on any items retrieved from the environment to access column values. Since the new production is not known to other extensions, any associated type representation information is not available to them. Other extensions may not retrieve or modify the SQL extension's bindings, thereby causing subtle unexpected behavior within the SQL extension's constructs. While it is true that an extension might improperly remove an element entirely from the environment, the error will probably be dramatic and quickly detected.

4.5 Designing the Host Language Type System for Composable Extensions

This section looks at the challenges involved in designing and implementing Java's type system in an extensible fashion. ableJ uses higher-order tree values to represent Java types and to perform operations on them. The ableJ specification provides hooks for extensions to interact with the host language's type system.

- Extensions may examine the resolved types of expressions and perform operations such as sub-typing checks to ensure that expressions are correctly typed.
- Extensions may also add new types, say for complex numbers. They may add concrete syntax for variables to be declared with the new type, and for new expression

```
grammar edu:umn:cs:melt:ablej;

nonterminal TypeRep with name;

- Retrieving a type's representation by name.
function getTypeRep TypeRep ::= name::String { ... }

- Checking if a type is a sub-type of another type.
function subTypeCheck Boolean ::= ltype::TypeRep rtype::TypeRep { ... }

- Checking if two types are the same.
function match Boolean ::= ltype::TypeRep rtype::TypeRep { ... }

abstract production double_type_rep
typerep::TypeRep ::= {
     typerep.name = "double";
}

abstract production array_type_rep
typerep::TypeRep ::= component::TypeRep {
     typerep.name = "array";
}
```

Figure 4.11: ableJ definitions for representing type information.

literals.

- Extensions may specify how their types interact with existing types, say by defining new sub-typing relations.
- They may overload existing operators to handle operands in the new type. If values are to be automatically type-converted in contexts such as assignments, mechanisms for automatic coercion may also be specified.

4.5.1 Extensible Type Representation via Higher-Order Attributes

Figure 4.11 shows some of the definitions in the ableJ host language for representing type information. This information is stored as tree values whose roots are labeled with the non-terminal symbol `TypeRep`. `TypeRep` trees are created from resolved type expressions (in which all identifiers have been associated with declarations) via abstract

```
grammar edu:umn:cs:melt:ablej:extensions:complex;
import edu:umn:cs:melt:ablej;

abstract production complex_type_rep
typerep::TypeRep ::= { ... }

concrete production complex_type
type::Type ::= 'complex' {
     type.typeRep = complex_type_rep ();
}
```

Figure 4.12: An example of an extension (complex numbers) that uses the ABLEJ definitions in Figure 4.11 to add a new type.

productions such as `double_type_rep` and `array_type_rep` which take type-specific information as children. The production `array_type_rep` takes the `TypeRep` representing the array element type as a child while the `double_type_rep` production takes no children. Extensions may add new types. The complex type extension constructs the complex type's internal representation using the abstract production `complex_type_rep`, as shown in Figure 4.12.

Non-terminals such as `Type` and `Expr` that have associated type information are decorated with the attribute `typeRep` of type `TypeRep`, as shown in the `double_type` and `reference` productions in Figure 4.13. This attribute is used during type-checking to retrieve and analyse expression types.

Type-specific information can be stored on `TypeRep` nodes using extra attributes, such as the synthesized attribute `component_type_rep` on array types. During type-checking, this information could be retrieved from `typerep` values using a set of "flag" attributes on `TypeRep`. However, we have implemented an extension to the Silver specification language that performs pattern-matching on tree values using production names [25]. This allows this information to be accessed in a much more convenient and functional-style manner. Thus the `while` production uses pattern-matching (and the `case` construct) to verify that its conditional expression is a boolean value, as shown in Figure 4.13. The enhanced `for` extension also uses pattern-matching on the collection's type representation to generate appropriate host language code, as shown in

```
grammar edu:umn:cs:melt:ablej;

synthesized attribute typeRep :: TypeRep occurs on Type, Expr;

- Types and expressions are decorated with the attribute typeRep.
concrete production double_type
type::Type ::= 'double' {
     type.typeRep = double_type_rep ();
}

concrete production reference
expr::Expr ::= id::Id_t {
     expr.typeRep = ...; - Extracted from the environment.
}

- Using pattern-matching on typerep during type-checking.
concrete production while
loop::Stmt ::= 'while' '(' cond:Expr ')' body::Stmt {
     loop.errors = (case cond.typeRep of
                        boolean_type_rep () => [ ]
                        | _ => [ "While condition must be boolean." ]
                    end) ++ cond.errors ++ body.errors;
}
```

Figure 4.13: ABLEJ definitions for performing type-checking and an example of their use.

Figure 4.14. The kind of iteration is determined in the `for_each` production via the local attributes `isCollection` and `isArray`, as shown in Figure 4.4. `isArray` is true if pattern-matching on the loop variable's type succeeds against the `array_type_rep` production. `isCollection` is set to the result of a call to `subTypeCheck` on the loop variable type and the `Collection` interface type. The latter is retrieved using the helper function `getTypeRep`.

4.5.2 Adding Sub-Types and Run-Time Conversions

Sub-typing relationships are implemented using the non-terminals and attributes shown in Figure 4.15. The non-terminal `SuperTypeInfo` is decorated with attributes that store

```
grammar edu:umn:cs:melt:ablej:extensions:foreach;
import edu:umn:cs:melt:ablej;

concrete production for_each
foreach::Stmt::= 'for' '('type::Type var::Id':' group::Expr')' body::Stmt{

   - Using functions defined in the host language to perform type-checking.
   local attribute isCollection :: Boolean;
   isCollection = subTypeCheck (group.typeRep, getTypeRep ("Collection"));

   - Using pattern-matching on typerep during type-checking.
   local attribute isArray :: Boolean;
   isArray = case group.typeRep of array_type_rep (_) => true
                                  | _ => false
             end;
}
```

Figure 4.14: An example of an extension (the enhanced `for` loop) that uses the ableJ definitions in Figure 4.13 to perform type-checking.

the sub-type's and the super-type's `TypeRep`, and perform run-time conversion from expressions in the sub-type to corresponding ones in the super-type. Each `TypeRep` has an attribute `superTypes` that is a list of `SuperTypeInfo` elements. Each of these elements encapsulates a particular sub-typing relation. `superTypes` is a collection attribute and extensions can therefore augment its host language-initialized value with their own contributions via aspect productions.

For example, the complex type extension adds a new sub-typing relation. Since the Java double type has no super-types, the host language's `double_type_rep` production initializes its `superTypes` to the empty list `[ ]`, as shown in Figure 4.16. Since the complex type is a super-type of this type, the complex extension defines an aspect production to `double_type_rep` that adds an element to its `superTypes` attribute. The element is built from the `double_to_complex` production. The definition of the synthesized attribute `convertedExpr` in `double_to_complex` specifies how double-typed expressions are to be automatically coerced to equivalent complex expressions in contexts such as assignments. The converted expressions are constructed as complex literals with real parts set to the double values and the imaginary parts set to the double literal `0.0`. When

```
grammar edu:umn:cs:melt:ablej;

synthesized attribute subType :: TypeRep;
synthesized attribute superType :: TypeRep;
inherited   attribute exprToConvert :: Expr;
synthesized attribute convertedExpr :: Expr;

nonterminal SuperTypeInfo with
     subType, superType, exprToConvert, convertedExpr;
synthesized attribute superTypes :: [ SuperTypeInfo ] collect with ++
     occurs on TypeRep;

abstract production double_type_rep
typerep::TypeRep ::= {
     typerep.superTypes := [ ];
}

abstract production array_type_rep
typerep::TypeRep ::= component::TypeRep {
     typerep.superTypes := [ ];
}
```

Figure 4.15: ABLEJ definitions for specifying sub-typing relations.

no source-level conversion is required between types (say between two Java 1.4 classes), then the value of `res.convertedExpr` in the production defining the `SuperTypeInfo` will be the same as `res.exprToConvert`. The host language thus provides a mechanism by which the extensions can link new types such as complex numbers to the existing type hierarchy.

The sub-typing information in the `SuperTypeInfo` non-terminals can be used to perform sub-typing checks. This is done by calling the `sub_type_check` production (shown in Figure 4.17) with the two types as arguments. The result of this production call is of non-terminal type `SubTypeResult`, which has a boolean synthesized attribute `isSubType`. The value of this attribute is evaluated based on each type's `superTypes`. Run-time conversion is performed using the inherited attribute `exprToConvert` and synthesized attribute `convertedExpr`. Both of these attributes decorate `SuperTypeInfo` and `SubTypeResult`. `exprToConvert` is set to the input expression while `convertedExpr` is

```
grammar edu:umn:cs:melt:ablej:extensions:complex;
import edu:umn:cs:melt:ablej;

aspect production double_type_rep
typerep::TypeRep ::= {
     typerep.superTypes <- [ double_to_complex () ];
}

abstract production double_to_complex
res::SuperTypeInfo ::= {
     res.subType = double_type_rep ();
     res.superType = complex_type_rep ();
     res.convertedExpr = complex_literal ('complex',
                  '(', res.exprToConvert, ',', double_constant (0.0), ')');
}
```

Figure 4.16: An example of an extension (complex numbers) that uses the ableJ definitions in Figure 4.15 to specify new sub-typing relations.

the output type-converted expression. The fragment of code below shows how the host language grammar uses this facility in the `assignment` production.

```
local attribute subTypeResult :: SubTypeResult;
subTypeResult = sub_type_check (lhs.typeRep, rhs.typeRep);
subTypeResult.exprToConvert = rhs;
forwards to if subTypeResult.isSubType
            then converted_assignment (lhs, subTypeResult.convertedExpr);
            else converted_assignment (lhs, rhs)
```

The production checks if its left-hand side is a sub-type of its right-hand side. If it is, then the production forwards to a different production in which the right-hand side is replaced with an equivalent expression in the sub-type, generated using the run-time conversion facility returned by a call to `sub_type_check`. The run-time conversion is performed using a local attribute `subTypeResult`. As described in Section 2.2.1, local attributes are associated with specific productions, rather than with particular non-terminals. They evaluate to attributed trees whose roots can be given inherited attributes by the production on which they are defined. Here, the value of the inherited attribute `exprToConvert`

```
grammar edu:umn:cs:melt:ablej;
synthesized attribute isSubType :: Boolean;
nonterminal SubTypeResult with isSubType, exprToConvert, convertedExpr;

abstract production sub_type_check
result::SubTypeResult ::= lhs::TypeRep rhs::TypeRep {
     result.isSubType = ...; - Checks if lhs.superTypes includes rhs.
     result.convertedExpr = ...; - Type-converted result.exprToConvert.
}
```

Figure 4.17: ableJ definitions for performing sub-typing checks.

on `subTypeResult` is set to the assignment's right-hand side. Synthesized attributes on the roots of local attributes, such as the value of `convertedExpr`, can be referenced by any definition on the production on which they are defined. The complete `assignment` production (described in Section 4.5.3) incorporates this sub-typing check into a more general provision for operator overloading.

The sub-typing check that uses each type's `superTypes` assumes that there are no circularities in the type hierarchy. While the host language type hierarchy can be defined to be non-circular, it is possible that two extensions independently add sub-type relations that when composed, do create a circularity. This is a potential source of errors when composing two or more independently developed extensions which modify the type system. This problem would be avoided if every extension adhered to the condition that no extension type can be both a sub-type of a host language type, and a super-type of a host language type.

4.5.3 Performing Operator Overloading

Overloading operators allows programmers to write more concise and maintainable code. ableJ's specification uses collection attributes as a way to allow extensions to easily overload existing arithmetic and assignment productions. An example (shown in Figure 4.18) is the `transformed` attribute in the `assignment` production, which, in the absence of any errors, evaluates to a list containing a single type-specific assignment sub-tree. The `assignment` production forwards to this tree, and most attribute instances are retrieved

```
grammar edu:umn:cs:melt:ablej;

concrete production assignment
assign::Stmt ::= lhs::Expr '=' rhs::Expr ';' {

  production attribute transformed :: [ Stmt ] collect with ++;
  assign.transformed := if subTypeResult.isSubType
         then [ converted_assignment (lhs, subTypeResult.convertedExpr) ]
         else [ ];

  assign.errors = if length (assign.transformed) == 1 then [ ]
    else if length (assign.transformed) == 0 then [ "Incompatible types." ]
    else [ "Internal error due to multiple translations." ];

  forwards to if length (assign.transformed) == 1
              then head (assign.transformed)
              else skip ();
}
```

Figure 4.18: ABLEJ code that uses collection attributes to define type-specific assignment sub-trees.

```
grammar edu:umn:cs:melt:ablej:extensions:autoboxing;
import edu:umn:cs:melt:ablej;

aspect production assignment
assign::Stmt ::= lhs::Expr '=' rhs::Expr ';' {

  assign.transformed <-
     if  match (lhs.typeRep, getTypeRep ("Integer"))
      && match (rhs.typeRep, int_type_rep ())
     then [ converted_assignment (lhs, ...) ]; -- "new Integer (rhs)"
     else [ ];
}
```

Figure 4.19: An example of an extension (auto-boxing) that uses collection attributes to extend the host language's semantics.

off it. This provides a way to specify a specialized overloaded version of the assignment statement as extensions can contribute elements to this collection attribute via aspect productions. The auto-boxing extension uses this mechanism. As shown in Figure 4.19, when its left-hand side is of type `Integer` and its right-hand side is a primitive integer, the `assignment` production forwards to a version in which the right-hand side is a boxed integer.

The assignment production specification is more complex than it would be if it were not designed for composability; we want the extensions to be easy to write. The ableJ specification includes similar productions for other value-copying operations such as parameter passing, which can therefore be overloaded in a similar fashion. Providing the operation overloading idiom from within Silver's expressive specification language allows extensions such as the computational geometry extension [30] to specify optimizations for entirely new types. This distinguishes Silver from overloading in languages such as C++.

At Silver run-time (i.e., during compilation of code in the extended language), every forwarded tree must be defined by a single value. For type-specific dispatch to work, there must be exactly one element in the evaluated value of collection attributes such as `transformed`. However, `transformed` may have multiple elements if two extensions independently specialize an operation such as `assignment` under overlapping pre-conditions.

```
data List = Nil
          | Cons char List

append :: List -> List -> List
append Nil ys = ys
append (Cons head tail) ys = Cons head (append tail ys)

main :: IO ()
main = putStr l3
       where l1 = Cons 'a' Nil
             l2 = Cons 'b' (Cons 'c' Nil)
             l3 = append l1 l2
```

Figure 4.20: Defining an algebraic data-type for lists in Haskell.

Silver's collection attributes do not perform compile-time checking to verify that attributes such as `transformed` have only one element. Such errors are only detected at run-time during compilation of extended source code, as shown in Figure 4.18.

4.6 Other Extensions to ableJ

The ableJ host language grammar has been extended in ways other than those described so far. In this section, we give brief descriptions of three of these extensions. The first was implemented by the author, the second by other members of the MELT group. The list of ableJ extensions described in this chapter is not exhaustive. Others include:

- an extension to perform dimension analysis [31],
- an extension that adds notations for modeling languages, such as condition tables from the modeling language RSML^{-e}, that help in specifying complex boolean conditions [32, 33], and
- an extension that extends Java 1.4 with constructs for programming in the domain of computational geometry [30].

4.6.1 Adding Algebraic Data-types and Pattern-Matching

This section describes a general-purpose extension to Java that adds Pizza-style [34] constructs for algebraic data-types with pattern-matching. This allows for programming in Java using functional idioms, such as those in the Haskell program shown in Figure 4.20. This program defines an algebraic data-type `List` with two constructors: `Nil` that constructs the empty list and `Cons (head, tail)` that constructs non-empty lists. The function `append` on lists has two cases based on the pattern of the first argument.

Corresponding code in a version of Java extended with data-types is shown in Figure 4.21. It uses the `algebraic class` construct to declare a `List` data-type, and `case` member declarations to define the two types of lists. Patterns in the functional-style `append` method are declared within a modified `switch` statement. `case` patterns are built from constructor names and formal arguments that represent the list components. The switched expression is checked against each pattern in sequence. On a match, the actual list components are assigned to the `case` pattern's formal arguments and the corresponding statement is executed.

Valid equivalent Java 1.4 code is generated as with Pizza [34]. For each algebraic data-type, the extended compiler generates an abstract class with sub-classes corresponding to each case, as shown in Figure 4.22. Sub-classes have fields and constructor parameters that correspond to the types specified in the value constructors. For this example, the generated code includes an abstract `List` class with `Nil` and `Cons` sub-classes. The `Cons` sub-class has a `char` field for the list's head and a `List` field for the list's tail. Finally, the pattern-matching `switch` statement in the `append` method is translated to a nested `if` statement that performs instance checks to determine which constructor matches the `case` clause. Typed list components are retrieved using safe run-time type conversions.

4.6.2 Embedding the SQL Query Language

In this section we describe an extension that embeds the SQL query language into Java [35]. Figure 4.23 gives a fragment of code in this extended version of Java. A table `students` is defined using the new `import table` construct. The `students` table has three typed columns. The column `name` is of type `VARCHAR` (which is the type of SQL

```
algebraic class List {
    case Nil;
    case Cons (char, List);

    public List append (List ys) {
      switch (this) {
         case Nil: return ys;
         case Cons (head, tail): return new Cons (head, tail.append (ys));
      }
      return null;
    }

    public static void main (String[] args) {
      List l1 = new Cons ('a', new Nil ());
      List l2 = new Cons ('b', new Cons ('c', new Nil ()));
      List l3 = l1.append (l2);
      System.out.println (l3);
    }
}
```

Figure 4.21: A functional-style definition of a list data-type written in a version of Java extended with algebraic data-types.

strings). This type information allows for compile time detection of syntactic errors within queries, which are specified with the `using ... query` construct. For example, in the expression `age > max` if the variable `age` were of type `VARCHAR` and not `INTEGER`, this would be detected as a compile-time type error. The extension accesses and extends the symbol-tables defined by the host language to perform this compile-time domain-specific type-checking.

The extension specifies equivalent pure Java 1.4 code that uses the JDBC library, as shown in the second code fragment in Figure 4.23. Queries are constructed as Java `String`s, which are then passed to database servers for execution via a library call. The raw string queries are not checked at compile time. Thus the type error posited above would only be detected at run-time on the server when the query is executed. This is less safe than when queries are specified using the constructs provided by the SQL extension.

```java
abstract class List {
    public List append (List ys) {

        if (this instanceof Nil) {
            return ys;

        } else if (this instanceof Cons) {
            char head = (Cons)this._c_field_1;
            List tail = (Cons)this._L_field_2;
            return new Cons (head, tail.append (ys));
        }
        return null;
    }

    public static void main (String[] args) {
        List l1 = new Cons ('a', new Nil ());
        List l2 = new Cons ('b', new Cons ('c', new Nil ()));
        List l3 = l1.append (l2);
        System.out.println (l3);
    }
}

class Nil extends List {
    Nil () { }
}

class Cons extends List {
    private char _c_field_1;
    private List _L_field_2;
    List (final char _c_field_1, final List _L_field_2) {
        this._c_field_1 = _c_field_1;
        this._L_field_2 = _L_field_2;
    }
}
```

Figure 4.22: The equivalent pure Java 1.4 code that is generated for the extended code in Figure 4.21.

```
public class Students {
  public static void main (String args) {
    import table students [ id INTEGER, name VARCHAR, age INTEGER ];
    int max = 30;
    connection conn = "jdbc:/db/studentdb";
    ResultSet rs = using conn query
                    { SELECT name FROM students WHERE age > max };
  }
}
```

```
public class Students {
  public static void main (String args) {
    int max = 30;
    Connection conn = DriverManager.getConnection("jdbc:/db/studentdb");
    ResultSet rs = conn.createStatement().executeQuery(("SELECT " + "name"
      + " FROM " + "students" + " WHERE " + "age" + " > " + max ));
   }
}
```

Figure 4.23: Code written in a version of Java embedded with the SQL query language, and equivalent Java 1.4 code.

4.7 Discussion

As described in Chapter 1, programmers could ideally build customized versions of their languages with desired domain abstractions, with no implementation-level knowledge of the host language or the extensions, or expertise in compilers and programming languages. But when developing non-trivial extensions, there are challenges with respect to how the extensions may access and modify host semantic analyses (such as types, environment and error-reporting) without interfering with other extensions doing the same. In this chapter, we showed how to design non-trivial host languages and language extensions for easy extensibility and safe composability so that the generated compilers act as expected, while allowing extension writers to interact closely with the host language type system and environment. Further, extension writers have to follow certain restrictions (that informally specify invariants on the composed language) to ensure that their extensions work with other well-behaved extensions.

With ableJ, we have extended the Java 1.4 with new statements, expressions and types with the same look-and-feel as constructs in the host language, and which fit comfortably with the host language syntax. Extensions range from a simple macro-like enhanced `for` extension, to more interesting ones for complex number types, auto-boxing and algebraic data-types with pattern-matching. The generated compilers provide appropriate errors in terms of the higher-level constructs used by the programmer, instead of reporting errors in terms of the generated host language translation (as with macros), thereby shielding programmers from having to analyse any generated Java code.

The ableJ host language specification has remained stable for the past few years, during which we have written several extensions, demonstrating that the host language design is practical and useful. Further evidence is provided by the fact that the host language specification has been extended by novice extension writers without much difficulty. In addition, most of the ableJ extensions are analysable for termination of higher-order attribute evaluation by the monolithic termination analysis described in Chapter 5. Some also pass the modular version of the analysis. Chapter 5 give more details of the termination analysis and the results of running it on the ableJ host and extension grammars.

Our experience with writing extensions allows us to make a few observations about the kinds of extensions that are appropriate to frameworks such as Silver. As attribute grammars are a syntax tree-driven formalism, this approach is well-suited to writing extensions that are primarily syntax-directed. Further, Silver and the notion of forwarding are particularly suited to implementing extensions whose semantics can be expressed in terms of equivalent host constructs, where the reduction from extended source code to equivalent host language source code is performed via local (not global) program transformations.

Silver is appropriate for situations where type or binding information can be specified in a declarative manner, say by using higher-order attributes to encapsulate information about the sub-typing relations. When semantic information is specified declaratively in this fashion, it is easier for extensions to augment them using additive mechanisms such as collection attributes. It also makes it easier to specify invariants in these situations that ensure composability, such as the invariant described in Section 4.5.2 that specifies how new sub-typing relations may be added to the ableJ host language type hierarchy.

There are extensions that are not handled well by Silver. There are problems when dealing with extensions where there is overlap between semantic values. An example is the situation where two or more extensions overload a particular operation (such as addition) for the same operand types via a mechanism such as that described in Section 4.5.3. The syntax tree-driven nature of the attribute grammar formalism also makes it difficult for Silver to handle semantic analyses that are primarily control-flow graph-related. An example of such an analysis is the Java definite assignment restriction that requires an identifier to be initialized before it is used. However, as a first step toward incorporating such analyses, we have written an extension (described in Appendix A) to the Silver specification language that extends it with constructs to perform data-flow analysis during attribute evaluation. Silver is similarly not well-suited to dealing with global program transformations.

The process of developing the ableJ host language and extensions has provided information that is transferable to other languages. Some of the information on writing composable semantics specifications is transferable to non-attribute grammar frameworks. For example the design of the type system and the invariants we have described to ensure composability is relevant to any declarative language specification framework.

To conclude, this work is a contribution toward the goal of a library model of language extensibility in which useful extension specifications can be automatically composed with a host language specification by application programmers with no compiler expertise, and without the need for any glue code.

Chapter 5

Analysing Attribute Grammars for Termination of Tree Creation

5.1 Introduction

In this chapter, we describe the second contribution of the dissertation, an attribute grammar analysis for non-terminating tree creation. In Chapter 1, we discussed the usefulness of a declarative framework for incremental extension of languages (similar to libraries). Further, our goal of extension composability implies we are interested in composable syntax and semantics specifications that can be used to automatically construct compilers for extended versions of languages. In Section 5.1.1, we look at the issue of non-terminating tree creation during the process of higher-order attribute evaluation, a possible reason for the non-termination of a compiler constructed by Silver. In Section 5.1.2, we define the problem as developing a static analysis on higher-order attribute grammars for non-terminating tree creation during attribute evaluation. We restrict our consideration to non-circular and complete grammars that have no non-terminating function calls. We also list criteria for success for any candidate solution. In Section 5.1.3, we give an overview of our solution, a static analysis that checks a given attribute grammar for certain restrictions that guarantee that no infinite tree creation sequences are possible during attribute evaluation. Section 5.1.4 gives an outline of the rest of the chapter which describes the analysis and the results of running it on the ableJ grammar.

5.1.1 Non-Terminating Tree Creation during Attribution

An issue when generating compilers in a higher-order attribute grammar framework such as Silver is the potential for improper termination of attribute evaluation during the execution of the generated compiler. As described in Section 2.3, in each evaluation step during attribute evaluation, an undefined evaluable attribute instance (i.e., an instance such that all the attribute instances its evaluation depends upon have been evaluated) is evaluated. This attribution process evaluates attribute instances one at a time until there are no undefined evaluable attribute instances. This can happen because

- all attribute instances have been evaluated, in which case evaluation is said to have terminated normally, or
- a particular attribute instance has no associated attribute definition, or
- there is a circularity in the dependencies between two attribute instances, so that neither can be evaluated before the other.

Abnormal termination of evaluation results because there are no evaluable instances even though there are undefined attribute instances, or because of function calls that terminate abnormally. Finally, there is the possibility that attribute evaluation may not terminate because the total number of attribute instances may not be finite. Every time a local higher-order attribute is evaluated, a new tree with new undefined attribute instances is created, possibly including new local attribute instances. Thus an infinite number of local trees may be created. For example, Figure 5.1 shows an attribute grammar (based on an example by Vogt *et al.* [17]) that is complete and has no circularities, but for which attribute evaluation does not terminate. The original program syntax tree $p_R(p_X())$ evaluates the local attribute l_A to a tree $p_A()$, which evaluates its local attribute l_X to a tree $p_X()$, which creates another local tree $p_A()$, and so on indefinitely. Another example of a grammar for which tree creation does not terminate during attribute evaluation is given in Appendix B.3.

We define an improper evaluation sequence as an evaluation sequence that either terminates abnormally, or does not terminate due to the creation of an infinite number of trees. Vogt *et al.* [17] extended the notion of well definedness to higher-order attribute grammars by listing the conditions for attribute evaluation to terminate normally, viz.

```
start nonterminal R;
nonterminal X, A;

concrete production pR r::R ::= x::X { }

concrete production pX x::X ::= {
  local attribute lA::A = pA ();
}

concrete production pA a::A ::= {
  local attribute lX::X = pX ();
}
```

Figure 5.1: G_0: a complete, non-circular grammar for which tree creation may not terminate during attribute evaluation.

- The grammar must be complete, i.e., every synthesized, inherited and local attribute instance on a node must have a definition that specifies how it is to be evaluated,
- The grammar must be non-circular, i.e., at every evaluation step, there must exist a partial order on all undefined attribute instances on all trees, such that if they are evaluated in this order, an attribute instance is evaluated only when all the attribute instances upon which it depends have also been evaluated, and
- Only a finite number of trees may be constructed in any evaluation sequence.

Existing static analyses check higher-order attribute grammars for non-circularity and completeness [16]. Vogt *et al.* [17] specify versions of the completeness and circularity tests for higher-order attribute grammars which guarantee that a given grammar is respectively, complete and non-circular. While they specify a sufficient condition for tree creation to be terminating, it does not appear to have been used in an actual implementation. If a grammar passes the circularity test, passes the completeness test and is such that all functions terminate normally, then its evaluation will not terminate abnormally. But if the grammar is higher-order, evaluation may be non-terminating. Thus the circularity and definedness tests are not sufficient to guarantee normal termination for higher-order attribute evaluation.

5.1.2 Problem: A Static Analysis for Tree Creation Termination

The problem explored in the second part of the dissertation is that of developing a static analysis for higher-order attribute grammars for non-terminating tree creation during attribute evaluation. We restrict our consideration to non-circular and complete grammars, with no non-terminating function calls. An analysis to guarantee that no evaluation sequences exist with infinite number of tree creation steps, combined with the completeness and circularity tests, would be sufficient to ensure normal termination of attribute evaluation.

Higher-order grammars without function calls (so-called "pure higher-order attribute grammars") are Turing-complete because they can implement single-taped Turing machines [36]. Thus the problem of whether higher-order attribute grammar evaluation terminates is undecidable even if all function calls terminate normally and the grammar is non-circular and complete. This distinguishes it from the problem of termination of canonical attribute grammar evaluation. Thus our challenge is to develop a sound and terminating procedure to detect higher-order termination. The nature of the problem requires us to walk a fine line between developing an analysis that is simple enough to be easily proven correct and efficiently executed, but is also useful enough to show termination for a large class of useful and interesting grammars. Since modular static analyses guaranteeing termination of generated compilers are especially useful in our extensible language framework, the question of what modular guarantees could be given when composing a set of grammars is also important.

5.1.2.1 Criteria for Success

- Since the problem of detecting non-termination in all cases is undecidable, our goal is to find a sound and terminating procedure.
- The analysis must be able to show termination of tree creation for non-trivial grammars such as ableJ and its extension specifications.
- The analysis must run on these grammars in a reasonable amount of time.
- We must be able to formally show correctness by proving that attribute evaluation for any grammar that passes the analysis always terminates normally.
- While a monolithic analysis would be useful, a modular analysis would be desirable

given our goal of a framework for composable language semantic specifications.

5.1.3 Solution: Using Non-Terminal Orderings and Rewrite Rules

We present a static analysis on higher-order attribute grammars that detects non-terminating tree creation during attribute evaluation. The analysis is conservative since there are grammars that are terminating but cannot be shown to be so by the analysis. But it is useful and can show termination for non-trivial grammars such as ableJ and some of its extensions in a reasonable amount of time. We analyse a specific class of higher-order attribute grammars, RHOAG, whose attribute definition expressions are a subset of the expressions described in Chapter 2. Further, we assume an evaluation model (described in Section 2.3) in which all attribute instances are evaluated. Finally, we assume that all grammars under consideration are complete and non-circular, with no non-terminating function calls. Under these assumptions, the problem of showing evaluation always terminates normally can be reduced to that of showing that all tree creation sequences terminate.

For a given evaluation sequence, a tree creation sequence is a sequence of trees starting from the original program syntax tree, in which each non-initial tree is created as the value of a local attribute on its predecessor. For example, for the grammar in Figure 5.1,

$$p_R(),\ p_A(),\ p_X(),\ p_A(),\ p_X()$$

is a tree creation sequence. More examples are given in Appendix B.1.1 and Appendix B.2.1. The observation that improper attribute evaluation implies the existence of an infinite tree creation sequence, follows from the fact that improper evaluation sequences are non-terminating. Further, all non-terminating evaluation sequences include an infinite number of local tree creation steps. Finally, for any evaluation sequence with an infinite number of local tree creation steps, we can construct an infinite tree creation sequence.

In RHOAG, the right-hand sides of the attribute definitions evaluated in tree creation steps are constructed from production symbols, terminal symbols, conditional expressions, signature tree variables, and attribute accesses. They do not contain function symbols. The construction of new tree term values using production symbols, terminal symbols and signature tree variables is similar to the process of term rewriting, for which there is extensive existing work on showing termination by ordering terms. But rewrite rules would not usefully model the use of inherited attribute access during tree construction, where evaluation requires the context of the defining production. We therefore distinguish between two kinds of steps: those that use inherited attributes, and all other ("non-inherited") tree creation steps which are amenable to modeling by rewrite rules.

The analysis first attempts to construct an ordering on the grammar's non-terminals, that ensures that in any tree creation sequence, there are only a finite number of "inherited" tree creation steps. If the ordering exists, then any infinite tree creation sequence contains an infinite sub-sequence consisting solely of non-inherited steps. The second part of the analysis generates a set of rewrite rules for each grammar that models local tree creation for non-inherited tree creation steps. For any tree creation sequence with no inherited attribute accesses, there exists a rewrite sequence of the same length. Thus if the rules terminate then no such sequence exists. Since the rules are simple, a simple production ordering may suffice to show termination of the generated rules. We therefore present a procedure that attempts to order the grammar productions so that productions on the right-hand sides of the generated rules are smaller than those on their left-hand sides. If the production ordering exists, then the rules are terminating. If the rules cannot be shown to be terminating via a simple production ordering, an alternative would be to use more powerful systems such as AProVE [37]. If the termination analysis succeeds in generating both the desired set of terminating rules and the desired non-terminal ordering, then we have a guarantee that no infinite tree creation sequences exist and all evaluation sequences terminate normally.

This analysis is monolithic. Our goal of composable semantics would be even better served by a modular termination analysis. We desire a static check on extension grammars that guarantees that the grammar resulting from composing the host language and any combination of extensions that pass this modular analysis, will pass the monolithic test above. We have developed such a modular version of the termination analysis described above. First, we provide a modular check on grammars that guarantees that the monolithic non-terminal ordering procedure will succeed on any combination of grammars that pass it. Second, we provide a modular check on grammars that guarantees that the monolithic production ordering procedure will succeed on any combination of grammars that pass it. If the production ordering is not sufficient to show termination for the host language or any extension, then the modular version is not useful, as a modular version of AProVE is not available. The modular analysis is not complete. There are extensions that compose properly but cannot be shown to do so. In Section 5.6.1, we present the results of running the modular termination test on ableJ's extensions. Since our monolithic analysis assumes definedness and non-circularity on grammars, modular versions of these analyses are also needed. Other members of the MELT group have developed such analyses [18].

5.1.4 Chapter Outline

- In Section 5.2, we define the syntax of attribute definitions and expressions in RHOAG, the class of higher-order attribute grammars handled by the analysis.
- In Section 5.3, we show how the problem of showing termination of higher-order attribution can in certain cases be reduced to that of disproving the existence of infinite tree creation sequences.
- In Section 5.4, we define a condition that orders the grammar's non-terminals so that any tree creation sequence contains only a finite number of steps that evaluate inherited attribute accesses. We further present a procedure that attempts to construct such an ordering for a given grammar. We also define a modular version of the procedure.
- In Section 5.5, we define a procedure that constructs a set of rewrite rules for a grammar so that if the rules are terminating, then there are no infinite tree creation sequences without inherited attributes. We present a simple procedure that attempts to show that the generated rules terminate by ordering the grammar's productions. We also define a modular version of the procedure.
- Section 5.6 concludes with a discussion of the analysis.

5.2 RHOAG: A Restricted Class of Higher-Order Attribute Grammars

The class RHOAG of higher-order attribute grammars handled by our termination analysis is a subset of the class defined in Section 2.2. The restrictions that define this class allow us to isolate the creation of new syntax trees during attribute evaluation to steps that evaluate local attributes. This lets us reduce the termination problem to that of disproving infinite tree creation sequences (as shown in Section 5.3). The restrictions are not overly restrictive. The resulting class includes most features available in Silver, and is expressive enough to specify grammars that are the equivalent of ABLEJ.

RHOAG is restricted in the following ways:

- Within the expressions that define attributes on a production, attribute values may only be accessed off child nodes or the roots of local attributes (in the case of synthesized attributes), or off the production's own node (in the case of inherited attributes). All other kinds of accesses, such as nested attribute accesses, are disallowed by the syntax. A production may not refer to synthesized attribute values on its own node, or inherited attribute values on its child nodes or the roots of its local attributes. Thus the set *Expr* of attribute definition expressions now has the form below, where $e_i \in Expr$.

  Expr ::=

	$\#i$		(parent or child tree)
$\mid$	$\#0.a_I$	$a_I \in A_I$	(inherited occurrence on parent)
$\mid$	$\#i.a_S$	$a_S \in A_S$	(synthesized occurrence on child)
$\mid$	$l.a_S$	$a_S \in A_S$	(synthesized occurrence on local)
$\mid$	$q(e_1,\ ...,\ e_{n_q})$	$q \in P$	(tree creation)
$\mid$	c	$c \in T$	(terminal symbol)
$\mid$	$f(e_1,\ ...,\ e_{n_f})$	$f \in F$	(function call)
$\mid$	`if` e_C `then` e_T `else` e_E		(conditional expression)

- In RHOAG, F is a set of primitive functions. They do not take trees as arguments or return them as results. Thus we now have the following:

$inTypes : F \longrightarrow (PT)*$ returns a function's input types.

$outType : F \longrightarrow PT$ returns a function's output type.

$applyFun : F \longrightarrow (PV)* \longrightarrow PV$ returns the result of applying a function to a list of values.

- Finally, for simplicity, there are no inherited attribute occurrences on the root node symbol. In other words, $\neg\, \exists a_I \,.\, a_I \in A_I \;\wedge\; a_I@S$.

Most Silver constructs not in RHOAG can be rewritten to equivalent ones in the class. Thus a Silver grammar can be translated to an equivalent one in RHOAG via a sound and terminating procedure.

- The prohibition against a production referencing synthesized attributes on its own node or inherited attributes on its children or the roots of its local attributes is not restrictive in a non-circular grammar as each production computes these values.
- The restriction that no inherited attributes may appear on the root node can be handled by adding new dummy non-terminals that exist only to pass inherited attributes to existing root symbols.
- Forwarding can be translated away using local attributes and extra attribute copying definitions.
- Silver functions can be easily translated to abstract productions in which the left-hand sides are non-terminals with special synthesized attributes that can be queried for return values. We will therefore not consider functions over tree values in our analysis. Tree expressions do not contain function symbols, except possibly in the flags of conditional expressions.

The attribute evaluation model (described in Section 2.3) is simpler than most standard evaluators, including Silver's. It does not attempt order attribute instances in order to evaluate only those attribute instances required to evaluate some specified attribute instance on the root. Since it evaluates more attribute instances than Silver, any attribute evaluation sequence possible in Silver for a given syntax tree is also included in this model. However, there are sequences in this model that are not possible in Silver. So there may be cases where a grammar's evaluation sequences are non-terminating in this model, while they are terminating in most standard evaluators such as Silver.

5.2.1 Example Grammars

We will use a few example grammars in our description of the termination analysis. A brief description of each grammar is given below. The complete Silver specifications of some of the grammars are lengthy and are given in Appendix B. The appendix also provides examples of valid programs in each grammar, along with examples of higher-order attribute evaluation, and the rules generated by the analysis. For some of the grammars, higher-order attribute evaluation always terminates. For others, non-terminating attribute evaluation sequences exist. The results of running the termination analysis on these grammars are given in Section 5.6.

G_0: This grammar (shown in Figure 5.1) was defined by Vogt *et al.* to illustrate the problem of non-terminating tree creation [17]. The rules generated by the analysis for this grammar are non-terminating. While our analysis cannot prove non-termination for an input grammar (it either detects termination, or else fails), the tool AProVE does flag G_0's rules as non-terminating.

G_1: This grammar is described in Appendix B.1 and specifies a small imperative language. It uses local attributes to provide a simple simulation of forwarding. It does not have any inherited attributes. Higher-order attribute evaluation always terminates for this grammar. Our analysis successfully detects this.

G_2: This grammar is described in Appendix B.2 and specifies a small imperative language. It uses multiple levels of inherited attribute occurrences to implement the program environment. There are inherited attributes that decorate non-terminals that are themselves the types of inherited attributes. Higher-order attribute evaluation always terminates for this grammar. Our analysis successfully detects this.

G_3: This grammar is described in Appendix B.3 and is an extended version of G_0. It defines additional attributes to construct a simple "host language translation" for each program in the grammar in which sub-trees are replaced with the trees they forward to. Higher-order evaluation is non-terminating for this grammar. While our analysis does not prove non-termination for an input grammar, the tool

AProVE does flag G_3's rules as non-terminating.

G_4: This grammar is shown in Figure 2.4. A Silver version is given in Chapter 3. It implements a simple let expression language. Higher-order attribute evaluation is non-terminating for this grammar. The non-terminal ordering phase of the analysis fails for G_4.

5.3 Reducing the Problem to Disproving Infinite Tree Creation Sequences

Theorem I: Given a non-circular, complete grammar in RHOAG with terminating functions, if there is an improper evaluation sequence for a tree in a grammar, then there is an infinite tree creation sequence for that tree.

For a given evaluation sequence, a tree creation sequence is a sequence of trees starting from the program syntax tree, in which each non-initial tree is the value of a local attribute that has been evaluated on its predecessor. Thus for an evaluation sequence for a syntax tree t_0 given by

$$\langle \mathcal{T}_0,\ \Gamma_0 \rangle \overset{n_0\#a_0}{\triangleright} \langle \mathcal{T}_1,\ \Gamma_1 \rangle \overset{n_1\#a_1}{\triangleright} \langle \mathcal{T}_2,\ \Gamma_2 \rangle \overset{n_2\#a_2}{\triangleright} ...$$

a tree creation sequence is a sequence of syntax trees $t_0,\ t_1,\ t_2, ...$ where there exist $k_0, k_1, k_2, ...$ where $0 \leq k_0 < k_1 < k_2 < ...$, that are indices of tree creating steps in the evaluation sequence, where for $i \leq 0$,

- $\langle \mathcal{T}_{k_i},\ \Gamma_{k_i} \rangle \overset{n_{k_i}\#a_{k_i}}{\triangleright}$
  $\langle \mathcal{T}_{k_i}\ \cup\ \{t_{i+1}\},\ \Gamma[n_{k_i}\#a_{k_i} \mapsto t_{i+1}]\ \cup\ \{[n'\#a' \mapsto \bot] \mid n'\#a' \in \mathit{instances}(t_{i+1})\}\rangle$,
- $n_{k_i}\#a_{k_i} \in \mathit{instances}(t_i)$,
- $a_{k_i} \in L$,
- $(a_{k_i} = e) \in \mathit{defs}(\mathit{prod}(n_{k_i}))$, and
- $t_{i+1} = \mathit{newTree}(\mathit{eval}(n_{k_i},\ e,\ \Gamma_{k_i}))$.

For example, for the grammar in Figure 5.1, $p_R()$, $p_A()$, $p_X()$, $p_A()$, $p_X()$ is a tree creation sequence. More examples of tree creation sequences are given in Appendix B.1.1 and Appendix B.2.1.

The reduction from improper evaluation sequences to infinite tree creation sequences follows from the fact that any improper evaluation sequence for a non-circular, complete grammar in RHOAG with terminating functions, is infinite and contains an infinite number of local tree creation steps, and for such a sequence, we can construct an infinite tree creation sequence. We can show this as follows:

- At every evaluation step for a non-circular grammar, there is an ordering in which all undefined attribute instances on all trees can be evaluated, so that each attribute instance is evaluated only after any attribute instances it requires have been defined. Thus evaluation terminates only if all attribute instances have been evaluated. Any improper evaluation sequences must therefore be non-terminating and must evaluate an infinite number of attribute instances.
- This implies that an infinite number of local trees are created. New trees with undefined attribute instances are only created in local tree creation steps. If the number of locals created were finite, the total number of attribute instances (given by the finite sum of each local's attribute instances and attribute instances in the original tree) would be finite. Thus an improper evaluation sequence contains an infinite number of tree creation steps.
- For such an evaluation sequence, we can construct an infinite tree creation sequence from the original tree t_0. If we cannot, t_0 has a finite number of local attribute instances. As each local attribute instance is evaluated exactly once, on a pre-existing tree, the evaluation of one of t_0's local attribute instances (say a_1) must result in the creation of an infinite number of local trees. Otherwise, the total number of trees created would be finite. This procedure can be repeated on a_1 to show that its evaluated tree value (say t_1) has a local attribute instance (say a_2) whose evaluation (to a tree t_2) results in the creation of an infinite number of local trees. This procedure can be continued indefinitely, resulting in the desired infinite tree creation sequence $t_0, t_1, t_2, ...$

Thus for any improper evaluation sequence, there is an infinite tree creation sequence from the original tree.

The third step above is a direct adaptation of the proof of König's Lemma, which states that an infinite tree in which each node has finitely many children, has an infinite path. We can organize each evaluation state's trees into a tree of locals (TOL), in which each node represents a syntax tree, and the root represents the original program syntax tree. A node's children represent the local trees created on any node in the syntax tree it represents. In the initial evaluation state, the TOL has one node, labeled with the program syntax tree. In every local evaluating step, a node for the new syntax tree is added to the TOL, as a child of the TOL node corresponding to the local's parent syntax tree (every local is evaluated only once and on an existing tree). A tree creation sequence at an evaluation step corresponds to a path in the TOL at that step. Examples of the generation of TOLs for particular evaluation sequences are given in Appendix B.1 and Appendix B.2. Thus for an evaluation sequence, we can construct a sequence of TOLs, in which each TOL has as many nodes as the number of trees created in the corresponding evaluation state. Each new TOL therefore has at least the same nodes and edges as its predecessor. In other words, the TOLs constructed for a given evaluation sequence are "non-shrinking". The number of children on each TOL node has a finite bound, as each syntax tree has a finite number of nodes, each with a finite number of locals. Thus by König's Lemma, there is an indefinitely increasing path in the TOLs, and a corresponding infinite tree creation sequence.

5.4 Limiting the Number of Inherited Access in Tree Creation Sequences

We now specify a condition on attribute grammars that ensures that for any infinite tree creation sequence, there exists one with only a finite number of trees created via inherited attribute accesses.

5.4.1 Ordering Non-Terminals to Limit Inherited Attribute Accesses

Suppose there exists a reflexive and transitive ordering $\succeq$ on the grammar's non-terminals that is well-founded (with respect to $\succ$) and satisfies the conditions below:

$\succeq$ is a relation on NT such that

1. $\succeq$ is reflexive, i.e., $X \succeq X$ for all $X \in NT$..
2. $\succeq$ is transitive, i.e., $X \succeq Y$ and $Y \succeq Z$ imply that $X \succeq Z$ for all $X, Y, Z \in NT$.
3. $\succ$ is well-founded, i.e., there is no infinite sequence $X_0 \succ X_1 \succ X_2 \succ ...$
4. $X \succeq Y$ if
   $(\exists p \in P \;.\; X = \mathit{lhs}(p),\; Y \in \mathit{rhs}(p)) \vee$
   $(\exists p \in P,\; l@p \;.\; X = \mathit{lhs}(p),\; Y = \mathit{type}_a(l)) \vee$
   $(\exists a_S \in A_S \;.\; a_S@X,\; Y = \mathit{type}_a(a_S))$.
5. $X \succ Y$ if $\exists a_I \in A_I \;.\; a_I@X,\; Y = \mathit{type}_a(a_I)$.

Figure 5.2: An ordering on a grammar's non-terminals that ensures that the trees in any tree creation sequence are non-increasing.

1. Non-terminal symbols are non-increasing from the root node of a syntax tree to its leaves. In other words, if there is a production with X as its left-hand side and a right-hand side containing Y, then $X \succeq Y$.
2. The root non-terminal of any tree value is no larger than the node on which it was evaluated. Thus, if a local attribute of type Y occurs on a production with left-hand side X, then $X \succeq Y$. Similarly, if a synthesized attribute of non-terminal type Y occurs on X, then $X \succeq Y$.
3. Finally, if an inherited attribute of non-terminal type Y occurs on X, then X is larger than Y. In other words, inherited occurs-on declarations are non-circular.

where

- $X \approx Y$ is the same as $(X \succeq Y \wedge Y \succeq X)$.
- $X \succ Y$ is the same as $(X \succeq Y \wedge \neg X \approx Y))$.
- $t_1 \succeq t_2$ is the same as $\mathit{symbol}(t_1) \succeq \mathit{symbol}(t_2)$. In other words, trees are compared using their root non-terminals.

Figure 5.2 gives a formal specification of the desired non-terminal ordering. Figure 5.3 shows a graphical representation of an ordering on the non-terminals of the grammar G_4 defined in Appendix B.2 that satisfies these conditions. Each node in this graph is labeled with one or more non-terminals. $X \succeq Y$ if and only if there is a (possibly empty) path from X's node to Y's node. Since the graph is acyclic, $X \approx Y$ if and only if X and Y label the same node. The ordering is derived from the grammar's definition and the graph's edges appear for different reasons. For example, `Stmt` appears on the right-hand side of a production (`root`) with `Root` as its left-hand side. On the other hand, `TypeEnv` is the type of an inherited attribute (`typeEnv`) that decorates `Env`. The four non-terminals at the top of the graph (`Root`, `Stmt`, `Expr` and `Type`) are used to construct the syntax tree. These are decorated by the three non-terminals at the bottom (`Env`, `TypeEnv` and `TypeRep`).

If such an ordering exists, then every tree created as a local attribute is no larger than the tree on which it was created. Thus the trees in any tree creation sequence (corresponding to a path in the tree of locals) are non-increasing. For example, consider the tree of locals constructed as shown in Figure B.8 for a tree in the grammar G_2. Consider the path $T_0,\ T_1,\ T_2,\ T_4$ and the corresponding tree creation sequence. If the ordering exists as described, we would have $T_0 \succeq T_1 \succeq T_2 \succeq T_4$. We state this formally as **Lemma** 1.

> **Lemma** 1: Assume $\succeq$ exists as described in Figure 5.2. For any evaluation sequence with a tree creation sequence $t_0,\ t_1,\ t_2, ...$ we have $t_i \succeq t_{i+1}$, for all $i \geq 0$.

Proof:

- Consider the elements t_i and t_{i+1} of the tree creation sequence $t_0,\ t_1,\ t_2, ...$
- There is a corresponding tree creation step in the evaluation sequence given by
  - $\langle \mathcal{T},\ \Gamma \rangle \overset{n\#a}{\rhd}$
    $\langle \mathcal{T} \cup \{t_{i+1}\},\ \Gamma[n\#a \mapsto t_{i+1}] \cup \{[n'\#a' \mapsto \bot] \mid n'\#a' \in \mathit{instances}(t_{i+1})\} \rangle$,
  - $n\#a \in \mathit{instances}(t_i)$,
  - $a \in L$,
  - $(a = e) \in \mathit{defs}(\mathit{prod}(n))$, and

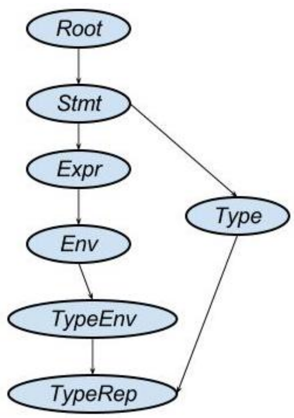


Figure 5.3: Ordering the non-terminals of grammar G_2 (defined in Appendix B.2) according to the restrictions in Figure 5.2. Each node is labeled with one or more non-terminals. $X \succeq Y$ if and only if there is a (possibly empty) path from X's node to Y's node.

  - $t_{i+1} = newTree(eval(n,\ e,\ \Gamma))$.

- For any edge $\langle n_j,\ n_{j+1}\rangle$ on the path of nodes from t_i's root node to the node n on which t_{i+1} is evaluated, we have $symbol(n_j) \succeq symbol(n_{j+1})$, since $lhs(prod(n_j)) \succeq X$, for all $X \in rhs(prod(n_j))$.
- Therefore, $symbol(t_i) \succeq symbol(n)$; i.e., in the parent tree t_i, non-terminals are non-increasing from the root node to the node on which the new tree is created.
- We also have $symbol(n) \succeq type_a(a) = type_e(prod(n),\ e) = symbol(t_{i+1})$.
- Therefore, $symbol(n) \succeq symbol(t_{i+1})$; i.e., the root symbol of the new tree is no larger than the symbol of the node on which it was created.

- Therefore, $symbol(t_i) \succeq symbol(n) \succeq symbol(t_{i+1})$, which means $t_i \succeq t_{i+1}$.

We define a constant tree creation sequences as a tree creation sequence with no steps in which the generated tree is strictly smaller (with respect to $\succeq$) than its parent tree. It is thus a tree creation sequence $\langle t_0,\ \Gamma_0\rangle, \langle t_1,\ \Gamma_1,\ n_1\#a_1\rangle, \langle t_2,\ \Gamma_2,\ n_2\#a_2\rangle, ...$ where $t_0 \approx t_1 \approx t_2 \approx ...$ For example, consider the path $T_0,\ T_1,\ T_2,\ T_4$ in the tree of locals constructed as described in Figure B.8, and the corresponding tree creation sequence. We have $T_0 \succeq T_1 \succeq T_2 \succeq T_4$. Under the ordering in Figure 5.3, we have $T_0 \succ T_1 \succ T_2 \approx T_4$. Here $T_2,\ T_4$ is a constant tree creation sequence.

Lemma 2: If $\succeq$ exists as described in Figure 5.2, then for any infinite tree creation sequence, there exists an infinite constant tree creation sequence.

Proof: The proof is by construction.

- Assume an infinite tree creation sequence
$\langle t_0,\ \Gamma_0\rangle, \langle t_1,\ \Gamma_1,\ n_1\#a_1\rangle, \langle t_2,\ \Gamma_2,\ n_2\#a_2\rangle, ...$
- By **Lemma** 1, the trees are non-increasing, so we have an infinite sequence $t_0 \succeq t_1 \succeq t_2 \succeq ...$
- We therefore have an infinite sequence $X_0 \succeq X_1 \succeq X_2 \succeq ...$ of non-terminals where $X_i = symbol(t_i)$.
- There is an $i \geq 0$ such that $t_j \approx t_{j+1}$ for all $j > i$, as otherwise, we have an infinite sequence $X_0 \succ X_1 \succ X_2 \succ ...$, which is not possible as $\succ$ is well-founded.
- We therefore have an infinite sequence $t_j \approx t_{j+1} \approx t_{j+2} \approx ...$ which is the desired infinite constant tree creation sequence.

5.4.2 Constructing the Desired Non-Terminal Ordering

We specify a sound and terminating procedure that, for a given attribute grammar, attempts to define an ordering on its non-terminals that satisfies the conditions in Figure 5.2. We introduce the following abbreviated notations to specify the conditions that the constructed ordering must satisfy.

- $S(X,\ Y) \equiv (\exists p \in P\ .\ X = lhs(p),\ Y \in rhs(p)) \vee$

Procedure A:

1. Construct a directed graph G_{NT} whose vertices correspond to the non-terminals in NT, and with an edge $\langle X,\ Y\rangle$ if and only if $S(X,\ Y)$.

2. Construct a directed graph G^{SCC} whose vertices correspond to the strongly connected components (SCC) of G_{NT}, and with an edge $\langle S_X,\ S_Y\rangle$ if and only if there exist $X \in S_X$ and $Y \in S_Y$ where either $I(X,\ Y)$, or $S(X,\ Y)$ and $\neg S(Y,\ X)$.

3. If G^{SCC} has cycles, return failure.

4. The desired ordering is defined as follows: $X \succeq Y$ if and only if there is a (possibly empty) path in G^{SCC} from X's SCC to Y's SCC.

Figure 5.4: A procedure that, for a given attribute grammar, attempts to define an ordering on its non-terminals that satisfies the conditions in Figure 5.2.

$(\exists p \in P,\ l@p\ .\ X = lhs(p),\ Y = type_a(l)) \vee$
$(\exists a_S \in A_S\ .\ a_S@X,\ Y = type_a(a_S))$.

- $I(X,\ Y)\ \equiv\ \exists a_I \in A_I\ .\ a_I@X,\ Y = type_a(a_I)$.

We need a procedure that constructs a reflexive and transitive relation $\succeq$ that is well-founded (with respect to $\succ$) on a given attribute grammar's non-terminals so that for any $X, Y \in NT$, $S(X,\ Y)$ implies $X \succeq Y$ and $I(X,\ Y)$ implies $X \succ Y$. **Procedure** A in Figure 5.4 gives such a procedure.

The intuition behind **Procedure** A can be understood as follows. For any tree creation sequence, we can construct a path in G^{SCC} of the nodes that correspond to the strongly connected components (SCC) of the non-terminal symbols at the roots of the trees in the sequence. For "non-inherited" tree creation steps, we either stay at the same SCC node (in those cases in which $X \approx Y$), or move to another SCC node (in those cases in which $X \succ Y$). For all other tree creation steps, we move to another SCC node. Thus if the graph has no cycles, then there can only a finite number of "inherited" tree creation steps.

Procedure A can be used to construct the ordering in Figure 5.3 for grammar G_2 defined in Appendix B.2. An example of a grammar for which the procedure does not succeed is grammar G_4 defined in Figure 2.4. G_4's non-terminals cannot be ordered as desired because the `Expr` non-terminal is decorated with an inherited attribute (`env`) of non-terminal type `Env`, one of whose productions (`consEnv`) contains `Expr` in its right-hand side.

Any ordering constructed by **Procedure** A satisfies the conditions above. We state this formally as **Lemma** 3.

> **Lemma** 3: The ordering $\succeq$ generated by **Procedure** A for a given attribute grammar satisfies the conditions in Figure 5.2.

Proof

1. To show: $\succeq$ is reflexive, i.e., $X \succeq X$ for all $X \in NT$.
   - We trivially have $X \succeq X$.
2. To show: $\succeq$ is transitive, i.e., $X \succeq Y$ and $Y \succeq Z$ imply that $X \succeq Z$, for all $X, Y, Z \in NT$.
   - Let $X \succeq Y$ and $Y \succeq Z$.
   - In G^{SCC}, there is a path from X's SCC to Y's SCC, and from Y's SCC to Z's SCC.
   - This means there is a path from X's SCC to Z's SCC, and so $X \succeq Z$.
3. To show: $\succ$ is well-founded, i.e., there is no infinite sequence of non-terminals $X_0 \succ X_1 \succ X_2 \succ ...$
   - Assume $\succ$ is not well-founded and there is an infinite sequence of non-terminals $X_0 \succ X_1 \succ X_2 \succ ...$
   - If there are any repetitions of a non-terminal X_i in this sequence, we can construct a cyclical path in G^{SCC} containing X_i's SCC, which is a contradiction.
   - This means that there are no repetitions in the sequence, and an infinite number of distinct non-terminals, which is a contradiction since NT is finite.
4. To show: $S(X,\ Y)$ implies $X \succeq Y$.

- Case 1: If $S(Y,\ X)$, there are edges from X to Y and from Y to X in G_{NT}, and so X and Y are in the same SCC in G^{SCC}, and $X \succeq Y$.
- Case 2: If $\neg S(Y,\ X)$, then there is an edge from X's SCC to Y's SCC in G^{SCC}, and so $X \succeq Y$.

5. To show: $I(X,\ Y)$ implies $X \succ Y$

   - There is an edge from X's SCC to Y's SCC in G^{SCC} and so $X \succeq Y$, where $X \not\approx Y$ (as otherwise G^{SCC} has a cycle), which means $X \succ Y$.

5.4.3 A Modular Version of the Non-Terminal Ordering Procedure

We can specify a condition on extensions that guarantees that a valid non-terminal ordering exists for any combination of extensions that satisfy it. We also define a corresponding modular version of the monolithic non-terminal ordering procedure described in the previous section. Section 2.2.3 gives a short note on composing higher-order attribute grammars. We write $X \overset{G}{\succeq} Y$ if $X \succeq Y$ under the grammar G.

An extension passes the modular non-terminal ordering if it introduces no new ordering relation on host language non-terminals. In other words, for a host language grammar G_H with non-terminals NT_H, an extended grammar G_{H+E} passes the modular condition if and only if, for any two non-terminals $X, Y \in NT_H$,

$$(X \overset{G_{H+E}}{\succeq} Y \Longrightarrow X \overset{G_H}{\succeq} Y) \wedge (X \overset{G_{H+E}}{\succ} Y \Longrightarrow X \overset{G_H}{\succ} Y)$$

We state this formally as **Lemma** 4.

Lemma 4:

- Say the host language grammar $G_H =$
  $\langle\ \langle\ NT_H,\ T_H,\ P_H,\ S\ \rangle,\ PT,\ PV,\ F,\ A_S,\ A_I,\ L_H,\ defs_H\ \rangle$
  has an ordering $\overset{G_H}{\succeq}$ on NT_H that satisfies the conditions in Figure 5.2.

- Say there are n extended grammars where for $1 \leq i \leq n$, each extended grammar $G_{H+E_i} =$

$\langle \ \langle \ NT_H \ \cup \ NT_{E_i}, \ T_H \ \cup \ T_{E_i}, \ P_H \ \cup \ P_{E_i}, \ S \ \rangle, \ PT, \ PV, \ F,$
$A_{SH} \ \cup \ A_{SE_i}, \ A_{IH} \ \cup \ A_{IE_i}, \ L_H \ \cup \ L_{E_i}, \ \mathit{defs}_H \ \cup \ \mathit{defs}_{E_i} \ \rangle$
has an ordering $\overset{G_{H+E_i}}{\succeq}$ on $NT_H \ \cup \ NT_{E_i}$ that satisfies the conditions in Figure 5.2, where for $X, Y \in NT_H$,
$(X \overset{G_{H+E_i}}{\succeq} Y \Longrightarrow X \overset{G_H}{\succeq} Y) \ \wedge \ (X \overset{G_{H+E_i}}{\succ} Y \Longrightarrow X \overset{G_H}{\succ} Y).$

then the desired ordering satisfying the conditions in Figure 5.2 exists on the non-terminals $NT_H \ \cup \ NT_{E_1} \ \cup \ NT_{E_2} \ ... \ \cup \ NT_{E_n}$ of the host language composed with these extensions.

Proof:

- Let G_H^{SCC} be the SCC graph of the host language H.
- Let $G_{H+E_i}^{SCC}$ be the SCC graph of the extended grammar $H + E_i$.
- Let $G_{H+E_1+...+E_n}^{SCC}$ be the SCC graph of the host language composed with $E_1, ..., E_n$.
- We represent the SCC containing a non-terminal X in an SCC graph G as $\mathcal{V}(G, X)$.
- For any $X, Y \in NT_H$, if $\mathcal{V}(G_H^{SCC}, X) = \mathcal{V}(G_H^{SCC}, Y)$ then $\mathcal{V}(G_{H+E_i}^{SCC}, X) = \mathcal{V}(G_{H+E_i}^{SCC}, Y)$, for any extended grammar $H + E_i$.
- Further, for any $X, Y \in NT_H$, if $\mathcal{V}(G_{H+E_i}^{SCC}, X) = \mathcal{V}(G_{H+E_i}^{SCC}, Y)$, we have $X \overset{G_{H+E_i}}{\succeq} Y$ and $Y \overset{G_{H+E_i}}{\succeq} X$, which by the modularity condition implies $X \overset{G_H}{\succeq} Y$ and $Y \overset{G_H}{\succeq} X$, and so $\mathcal{V}(G_H^{SCC}, X) = \mathcal{V}(G_H^{SCC}, Y)$.
- Thus for any $X, Y \in NT_H$ we have $\mathcal{V}(G_H^{SCC}, X) = \mathcal{V}(G_H^{SCC}, Y)$ if and only if $\mathcal{V}(G_{H+E_i}^{SCC}, X) = \mathcal{V}(G_{H+E_i}^{SCC}, Y)$, for any extended grammar $H + E_i$.
- Therefore $G_{H+E_1+...+E_n}^{SCC}$ can be obtained by taking the union of the SCC graphs for each extension.
- Let $S_1^H, ..., S_{n_H}^H$ be the SCCs (vertices) of G_H^{SCC}.
- Let $S_1^{E_i}, ..., S_{n_{E_i}}^{E_i}$ be the SCCs (vertices) of $G_{E_i}^{SCC}$, for $1 \leq i \leq n$.
- The SCCs (vertices) of $G_{H+E_1+...+E_n}^{SCC}$ are given by the set
$\{S_j^H \ \cup \ \bigcup_{i=i}^{n} (S_j^{E_i}, \text{ where } 1 \leq k \leq n_{E_i}, \ S_j^H \subseteq S_k^{E_i}) \mid 1 \leq j \leq n_H\}$
$\cup \ \{S_j^{E_i} \mid S_j^{E_i} \subseteq NT_{E_i}, \ 1 \leq j \leq n_{E_i}, \ 1 \leq i \leq n \ \}$
i.e., the composed grammar's SCC graph includes one SCC for each of G_H^{SCC}'s

SCCs, which further contains any extension non-terminals that appear in the corresponding SCC in the extension's SCC graph, as well as one SCC for each SCC in each extension's SCC graph that contains only extension non-terminals.

- There is an edge in $G^{SCC}_{H+E_1+...+E_n}$ from S_1 to S_2 if
  - $X, Y \in NT_H, X \in S_1, Y \in S_2$, and there is an edge in G^{SCC}_H from $\mathcal{V}(G^{SCC}_H, X)$ to $\mathcal{V}(G^{SCC}_H, Y)$, or
  - $X, Y \in NT_{E_i}, X \in S_1, Y \in S_2$, and there is an edge in $G^{SCC}_{H+E_i}$ from $\mathcal{V}(G^{SCC}_{H+E_i}, X)$ to $\mathcal{V}(G^{SCC}_{H+E_i}, Y)$, for some extension E_i.
- We need to show that $G^{SCC}_{H+E_1+...+E_n}$ has no cycles.
- Assume there is a cycle in $G^{SCC}_{H+E_1+...+E_n}$.
- The cycle cannot consist solely of SCCs belonging to a single extension (since it passed the monolithic analysis), and there are no edges between SCCs containing only non-terminals from different extensions.
- Thus the cycle consists of sections bounded by SCCs containing host language non-terminals (and possibly extension non-terminals) such that all SCCs in between contain non-terminals from a single extension.
- Consider such a section $h_1, v_1, ..., v_n, h_2$ where $X \in h_1, Y \in h_2$ for two host non-terminals $X, Y \in NT_H$, and $v_1, ..., v_n$ are SCCs containing only non-terminals belonging to some extended grammar $H + E_i$.
- Since the corresponding path is present in $G^{SCC}_{H+E_i}$, we have $X \overset{G_{H+E_i}}{\succeq} Y$, and therefore $X \overset{G_H}{\succeq} Y$ by the modularity condition.
- We can repeat this for each of the sections in the cycle in $G^{SCC}_{H+E_1+...+E_n}$, so that we have a cycle in G^{SCC}_H, which is a contradiction (as we assume the host language passed the monolithic analysis).
- Therefore $G^{SCC}_{H+E_1+...+E_n}$ has no cycles and the composed grammar passes the monolithic analysis.

To check whether an extension E satisfies the modular ordering condition, we compare G^{SCC}_H and G^{SCC}_{H+E}, the host language H's and the extended grammar's SCC graphs, using the procedure in Figure 5.5. For each pair of host language non-terminals, we verify that they are in the same SCC in G^{SCC}_{H+E} only if they are in the same SCC in

Procedure B:

1. Construct G_H^{SCC} and G_{H+E}^{SCC}, the host language's and the extended grammar's SCC graphs, as described in **Procedure** A.
2. For each X in NT_H
   (a) For each Y in NT_H
      i. If X and Y are in the same SCC node in G_{H+E}^{SCC}, and if X and Y are *not* in the same SCC node in G_H^{SCC}, then return failure.
      ii. If there is a non-trivial path from X's SCC to Y's SCC in G_{H+E}^{SCC}, and if there is *no* path from X's SCC to Y's SCC in G_H^{SCC}, then return failure.
3. Return success.

Figure 5.5: A modular analysis that checks extension grammars for restrictions that ensure that **Procedure** A succeeds on any combination of this extension with other well-behaved extensions.

G_H^{SCC}, and check that there is a path between SCCs in G_{H+E}^{SCC} only if there is one in G_H^{SCC}. Note that this same principle can be used on extensions that depend on other extensions; in this case the "host language" would be the host language composed with the other required extensions.

5.5 Modeling Constant Tree Creation Sequences with Term Rewrite Rules

In this section, we describe a procedure that checks whether a given grammar satisfies certain conditions that ensure that no infinite constant tree creation sequences are possible during attribute evaluation. It generates a set of rewrite rules from the definitions of higher-order attributes in the grammar's productions. The rules are such that for any constant tree creation sequence, we can generate a rewrite sequence of the same length. Thus if the rules are terminating, then there are no constant tree creation sequences. Further, if the ordering on non-terminals described in the previous section exists, then no non-terminating evaluation sequences exist for the grammar.

We also describe a procedure that attempts to analyse whether the rules terminate, via a simple ordering on the grammar's productions, based on how they occur in the generated rules. Such a procedure is sufficient to show termination of the rules generated for many grammars such as ableJ. We also provide a modular version of this production ordering procedure that can be run on extension grammars to ensure composability with other well-behaved extensions. For grammars for which this production ordering procedure cannot show rule termination, we analyse the generated rules for termination using more powerful tools such as AProVE.

5.5.1 Using Rewrite Rules to Model Constant Tree Creation Steps

In this section we describe the structure of the rewrite rules we use to model constant tree creation steps and sequences. The rules are such that for any constant tree creation sequence, we can generate a rewrite sequence of the same length. Each generated rule is associated with a specific production and can be applied to tree values (of type *Term*) with that production at the root, to derive values that correspond to tree values evaluated on the root node. Thus the rules derive each local in a constant tree creation sequence from its predecessor tree.

Each rule's left-hand side is restricted and consists of a term constructor and a list of rewrite variables. It is derived from its production's signature. The terms on the right-hand sides of rules have the same structure as tree values (of type *Term*), except for the following:

- They may refer to rewrite variables via numerical indices. This is similar to how the attribute definitions in Figure 2.3 refer to production signature tree variables via numerical indices.
- They represent inherited attribute accesses using the special term `INH`. This extra term is needed because rewrite rules do not incorporate the contextual information required to evaluate inherited attributes (such as attribute values on the parent node).

We formally represent the right-hand sides of these rewrite rules with the type *RuleRHS* which has the following definition:

$$
\begin{array}{lll}
\mathit{RuleRHS} ::= & & \\
 & c & \text{(terminal symbol)} \\
\mid & q(\mathit{RuleRHS}_1, ..., \mathit{RuleRHS}_{n_q}) & \text{(production symbol and sub-expressions)} \\
\mid & \#i & (i^{\texttt{th}} \text{ term variable}) \\
\mid & \texttt{INH} & \text{(inherited attribute access)}
\end{array}
$$

Note that $\mathit{Term} \subset \mathit{RuleRHS}$. We represent rules with the type

$$\mathit{Rule} \equiv P \times \mathit{RuleRHS}$$

For each production, a set of rules is generated based on its higher-order attribute definitions. The rules generated for the production p where $|rhs(p)| = n_p$ have the form $p(\#1, ..., \#n_p) \longrightarrow r$. Each rule has a different right-hand side based on the right-hand side of the higher-order attribute definition it is generated from. It further rewrites any tree term with this production on the root, to a tree constructed via the evaluation of that particular attribute definition.

The rules do not model tree creation steps with higher-order inherited attributes as they do not have `INH` on the left-hand side. However, they can *generate* `INH` terms. We therefore represent the elements of sequences derived via rewrite rules, with the distinct type *RTerm*, which has the following definition:

$$
\begin{array}{lll}
\mathit{RTerm} ::= & & \\
 & c & \text{(terminal symbol)} \\
\mid & q(\mathit{RTerm}_1, ..., \mathit{RTerm}_{n_q}) & \text{(production symbol and sub-expressions)} \\
\mid & \texttt{INH} & \text{(inherited access)}
\end{array}
$$

Note that $\mathit{Term} \subset \mathit{RTerm} \subset \mathit{RuleRHS}$. We define the following rewriting operators:

We write $t \xrightarrow{r} s$ if applying a rule given its right-hand side r on a tree term t results in the tree term s.

$\rightharpoondown \subseteq \mathit{RTerm} \times \mathit{RuleRHS} \times \mathit{RTerm}$

- $t \xrightarrow{\texttt{INH}} \texttt{INH}$.
- $p(t_1, ..., t_{n_p}) \xrightarrow{\#i} t_i$ where $1 \leq i \leq |rhs(p)|$.

- $t \xrightarrow{c} c$.
- $t \xrightarrow{q(r_1,...,r_{n_q})} q(s_1, ..., s_{n_q})$ where $t \xrightarrow{r_1} s_1, ..., t \xrightarrow{r_{n_p}} s_{n_p}$.

We write $t \stackrel{\mathcal{R}}{\Longrightarrow} s$ if the term t rewrites to s via a rule in the set $\mathcal{R}$ of rewrite rules.
$\stackrel{\mathcal{R}}{\Longrightarrow} \subseteq \ \mathit{RTerm} \ \times \ \mathit{RTerm}$

- $p(t_1, ..., t_{n_p}) \stackrel{\mathcal{R}}{\Longrightarrow} s$ if $p(t_1, ..., t_{n_p}) \xrightarrow{r} s$ for some $p(\#1, ..., \#n_p) \longrightarrow r \in \mathcal{R}$.
- $t \stackrel{\mathcal{R}}{\Longrightarrow} t[s_1 \rightsquigarrow s_2]$ if $s_1 \stackrel{\mathcal{R}}{\Longrightarrow} s_2$.

where $t[s_1 \rightsquigarrow s_2]$ is the result of replacing t's sub-term s_1 with the term s_2.
$[\rightsquigarrow] : \mathit{RTerm} \longrightarrow \mathit{RTerm} \longrightarrow \mathit{RTerm} \longrightarrow \mathit{RTerm}$

We define two additional short-hand rewriting operators:

- $t \stackrel{\mathcal{R}}{\Longrightarrow}^{*} s$ if $t \stackrel{\mathcal{R}}{\Longrightarrow} r_1 \stackrel{\mathcal{R}}{\Longrightarrow} ... \stackrel{\mathcal{R}}{\Longrightarrow} r_n \stackrel{\mathcal{R}}{\Longrightarrow} s, \ n \geq 0$.
- $t \stackrel{\mathcal{R}}{\Longrightarrow}^{+} s$ if $t \stackrel{\mathcal{R}}{\Longrightarrow} r_1 \stackrel{\mathcal{R}}{\Longrightarrow} ... \stackrel{\mathcal{R}}{\Longrightarrow} r_n \stackrel{\mathcal{R}}{\Longrightarrow} s, \ n > 0$.

5.5.2 Constructing the Rules for a Given Grammar

For a grammar with a set P of productions, the set of rewrite rules generated from its definitions is given by the function *getRules*(P) defined in Figure 5.1. It uses the function *ruleRHSs* (also defined in Figure 5.6) which returns the rules for a given higher-order definition. For example, for the grammar in Figure 5.1, the rules generated are

$$\{ \ p_X \longrightarrow p_A, \ p_A \longrightarrow p_X \ \}$$

This set of rules is clearly non-terminating. Other examples of the rules that are generated for specific grammars are given in Appendix B.1.2, Appendix B.2.2 and Appendix B.3.1.

The generated rules have the following characteristics:

- Explicit tree creation sub-expressions containing production names, terminal symbols and signature variables are represented as such in the rules' right hand sides.
- Conditional expressions are handled by generating separate rules for each sub-expression. Multiple rules may therefore be generated for a single definition or sub-expression.

$ruleRHSs : P \longrightarrow Expr \longrightarrow 2^{RuleRHS}$ returns the right-hand sides of the rules that model the evaluation of a given higher-order expression.

For an expression e on a production p where $type_e(p,\ e) \in (NT\ \cup\ T)$, $ruleRHSs(p,\ e)$ is defined as follows:

$ruleRHSs(p,\ \#i) = \{\#i\}$	($i^{\texttt{th}}$ sub-term)
$ruleRHSs(p,\ \#0.a_I) = \{\texttt{INH}\}$	(inherited access)
$ruleRHSs(p,\ \#i.a_S) = \{\#i\}$	($i^{\texttt{th}}$ sub-term)
$ruleRHSs(p,\ l.a_S) = ruleRHSs(p,\ e_L)$ where $(l = e_L) \in defs(p)$	(derived from local's definition)
$ruleRHSs(p,\ q(e_1, ..., e_{n_q})) = \{q(r_1, ..., r_{n_q}) \mid r_i \in ruleRHSs(p,\ e_i),\ 1 \leq i \leq n_q\}$	(rules for sub-rule permutations)
$ruleRHSs(p,\ c) = \{c\}$	(terminal symbol)
$ruleRHSs(p,\ \texttt{if}\ e_C\ \texttt{then}\ e_T\ \texttt{else}\ e_E) = ruleRHSs(p,\ e_T)\ \cup\ ruleRHSs(p,\ e_E)$	(rules for both sub-expressions)

$getRules : \mathcal{P}(P) \rightarrow \mathcal{P}(Rule)$ returns the set of rewrite rules that model attribute evaluation for a given set of grammar productions.

$$getRules(P) = \{\ p(\#1, ..., \#n_p) \longrightarrow r \mid p \in P,\ |rhs(p)| = n_p,\ r \in ruleRHSs(p,\ e),\ (x = e) \in defs(p),\ type_e(p,\ e) \in (NT\ \cup\ T)\ \}$$

Figure 5.6: Defining the rewrite rules that model higher-order attribute evaluation for a given set of productions.

- Accesses to synthesized attribute occurrences on a child are represented by the child's signature variable.
- Accesses to synthesized attribute occurrences on local attributes are represented by rewrite sub-terms generated from the locals' definitions. Local attributes are thereby in-lined when generating rewrite rules; this process terminates for non-circular attribute grammars. This process of in-lining during rule generation is distinct from the in-lining done at the grammar level, of synthesized attributes on parent nodes and inherited attributes on child nodes, to satisfy the format of definitions in RHOAG.
- Function symbols are not present in higher-order sub-expressions.

The rules thus retain production names and the structure of higher-order values, but do not keep track of the specific attribute instances that are accessed in each tree-creating expression. They abstract away local attributes, conditional expressions and specific attribute instance names, to generate a much simpler, albeit approximate model of the tree creation process. They do not model tree evaluation using inherited attributes. That part of the problem is handled by the non-terminal ordering procedure described in Section 5.4.

Since attribute instance names are not retained in the rewrite sub-terms corresponding to synthesized attribute accesses, the generated rules derive sequences corresponding to the evaluation of every possible higher-order synthesized instance on the child or in-lined local attribute instance. Only one of these will correspond to the actual run-time evaluation sequence in which a particular attribute instance is evaluated. Similarly, only one of the rules generated for the conditional expressions in a definition will model the tree created once all conditions are evaluated. Thus the rewrite rules are conservative in how they model tree creation since they derive more terms than are actually generated during higher-order evaluation. Their usefulness is demonstrated by the fact that they can show termination of grammars such as ABLEJ in a reasonable amount of time.

For a Constant TCS there is a Corresponding Rewrite Sequence

The rules generated for a grammar are terminating only if all constant tree creation sequences during its evaluation terminate. For any constant tree creation sequence, the rules can derive a "pruned" version of each tree from the "pruned" version of its predecessor tree, where the pruned version of each tree has some of its sub-terms (those constructed via inherited access evaluation) replaced by the `INH` term. Further, we can combine the rewrite sequences that correspond to each individual step to generate a rewrite sequence that is as long as the entire constant tree creation sequence. Thus if the rules generated as described are terminating, and the non-terminal ordering exists, then there are no constant tree creation sequences, and therefore no improper evaluation sequences.

We now formally define our notion of pruned trees and the relation between each tree in a constant tree creation sequence, and its corresponding pruned rewrite sequence term, in which some sub-terms (those constructed via inherited attribute evaluation) are replaced by the `INH` term. The pruned trees in the rewrite sequences retain enough of the structure of the original trees for rewrites to be performed on them corresponding to any non-inherited tree creation steps. This means that there is never a pruned tree that is just an `INH`.

Assume that the symbols on the roots of the trees in a particular constant tree sequence are of the same size as the non-terminal K. For any r that is the corresponding rewrite sequence term for a tree t, we have $r \sqsubset_K t$ where the relation $\sqsubset_K$ is defined as follows: $r \sqsubset_K t$ if and only if

- $t \in T,\ r = t$, or
- $t = q(t_1, ..., t_{n_q}),\ r = q(r_1, ..., r_{n_q}),\ r_1 \sqsubset_K t_1, ..., r_{n_q} \sqsubset_K t_{n_q}$, or
- $\mathit{lhs}(q) \succ K,\ r = \texttt{INH}$

$\sqsubset_K$ is defined such that the root symbols of any `INH` sub-trees are greater than K. Further, all nodes in the original trees that are the same size as K are present in the pruned trees. Rewrite rules can therefore still be applied to such nodes. **Lemma** 5 follows directly from the definition of $\sqsubset_K$.

Lemma 5: If $\mathit{lhs}(q) \approx K$ then any $u \sqsubset_K q(t_1, ..., t_{n_q})$ is of the form $q(r_1, ..., r_{n_q})$ where $r_1 \sqsubset_K t_1, ..., r_{n_q} \sqsubset_K t_{n_q}$.

For each step in a constant tree creation sequence, we can derive a pruned version of the generated tree from a pruned version of its predecessor (if it is not `INH`) via a non-empty rewrite sequence. Further, the last term in the rewrite sequence is not `INH`, which means the rewriting process can continue. The intuition behind this can be understood by breaking the rewrite sequence into two parts: the first term, and the rest of the sequence.

The first term is a pruned partially evaluated term in which all conditions in the local's defining expression have been evaluated, but in which attribute instance accesses have not been computed, and are represented by the sub-trees on which they are to be evaluated. The rules model conditional expressions by generating rules for both clauses in each expression. The first term is derivable from the tree term of the local attribute instance's defining node, via the one rule that corresponds to the actual evaluated values of the flags in each conditional expression.

In the first term, attribute accesses off children or locals in the tree's defining expression are represented by the rewrite term of the child or local, respectively. Since the generated rules also model the evaluation of these attribute instances off the child or local tree terms, additional rewrites can be performed on these sub-terms to generate the actual evaluated attribute instance value. The rest of the sequence thus rewrites the pruned sub-trees of this first term on which synthesized attribute instances are defined, to pruned versions of their evaluated values. The fact that these additional rewrites can be performed to generate the correct values can be shown inductively on the structure of the defining expressions. The inductive assumption is that all previously evaluated tree values have corresponding valid rewrite sequences from their parent sub-trees to the computed values. Further, if the local's root non-terminal is of the same size as its predecessor tree, then the derived pruned term is not an `INH` term.

We formally state the correspondence between each step in a constant tree creation sequence and its corresponding rewrite sequence as **Lemma** 6.

Lemma 6: For any evaluable higher-order attribute instance $n\#a$ in a state $\langle \mathcal{T},\ \Gamma \rangle$ with defining expression e where $symbol(n) \approx K$ for some $K \in NT$, we can rewrite any u where $u \sqsubset_K term(n)$ to v where $v \sqsubset_K \Gamma[n\#a]$, via a non-empty rewrite sequence of rules in $getRules(P)$. Further, if a's non-terminal type is of the same size as K, then v is not `INH`.

At any state $\langle \mathcal{T},\ \Gamma \rangle$ where
- $t' \in \mathcal{T}$
- $n\#a \in instances(t')$ with defining expression e
- $\forall n'\#a' \in dep(n,\ e,\ \Gamma)\ .\ \Gamma[n'\#a'] \neq \bot$
- $symbol(n)\ \approx\ K \in NT$
- $u \sqsubset_K term(n)$

we have
- $u \overset{\mathcal{R}}{\Longrightarrow}^{+} v$ where $v \sqsubset_K \Gamma[n\#a]$ and $\mathcal{R} = getRules(P)$
- if $type_a(a)\ \approx\ K$ then $v \neq$ `INH`.

The formal proof of **Lemma** 6 is given in Appendix C.2.

Given that we can construct sequences that derive each tree from its predecessor in a constant tree creation step, we can construct a rewrite sequence that models the entire constant tree creation sequence. The sub-sequences that generate the pruned versions of each tree can be linked to generate a rewrite sequence for the whole constant tree creation sequence. Thus the rules are terminating only if there are no infinite constant tree creation sequences.

In a constant tree creation sequence, each successive tree is evaluated on a node of its predecessor. Thus the first term of the rewrite sequence that models this tree creation step (as defined in **Lemma** 6) is a sub-term of the predecessor tree. So the predecessor tree itself can be rewritten to a term in which the local's parent sub-tree is replaced by the local's value. This new term contains the new local as a sub-term, so the process can be continued for the next evaluated local. As the rewrite sequence corresponding to each local's evaluation step is non-empty, we can construct a rewrite sequence as long as the entire constant tree creation sequence. We state this formally as **Lemma** 7.

Lemma 7: For a constant tree creation sequence $\langle t_0, \Gamma_0 \rangle, \langle t_1, \Gamma_1, n_1\#a_1 \rangle$, $\langle t_2, \Gamma_2, n_2\#a_2 \rangle, \ldots$ where for $i > 0, symbol(t_{i-1}) \approx symbol(t_i) \approx K \in NT$, we can define a function $R : N \longrightarrow RTerm$ so that for $i > 0,\ R(t_{i-1}) \overset{\mathcal{R}}{\Longrightarrow}^{+} R(t_i)$ where

$\mathcal{R} = \mathit{getRules}(P)$. In other words, for a constant tree creation sequence, we can construct a rewrite sequence of the same length.

Proof: The proof is by construction. Let $\mathcal{R} = \mathit{getRules}(P)$. It is sufficient to define a function $R : N \longrightarrow \mathit{RTerm}$ such that for $i \geq 0$

- $R(t_i)$ has a sub-term u_i where $u_i \sqsubset_K \mathit{term}(t_i)$ and $u_i \neq \texttt{INH}$, i.e., there is a rewritable sub-term corresponding to the parent tree of the next local tree
- $R(t_{i-1}) \overset{\mathcal{R}}{\Longrightarrow}^{+} R(t_i)$ if $i > 0$

We define R inductively as follows:
$R(t_0)$ is defined to be $\mathit{term}(t_0)$; here the inductive invariant holds trivially.
Consider the case where $i > 0$.

- By the inductive assumption, there exists a sub-term u_i of $R(t_i)$ such that $u_i \sqsubset_K \mathit{term}(t_i)$, $u_i \neq \texttt{INH}$ and $R(t_{i-1}) \overset{\mathcal{R}}{\Longrightarrow}^{+} R(t_i)$.
- We need to define $R(t_{i+1})$ such that it has a sub-term u_{i+1} where $u_{i+1} \sqsubset_K \mathit{term}(t_{i+1})$, $u_{i+1} \neq \texttt{INH}$ and $R(t_i) \overset{\mathcal{R}}{\Longrightarrow}^{+} R(t_{i+1})$.
- t_{i+1} is evaluated on n_{i+1} as the value of the local attribute instance $n_{i+1}\#a_{i+1}$.
- Since $n_{i+1} \in \mathit{nodes}(t_i)$ and $\mathit{symbol}(n_{i+1}) \approx \mathit{symbol}(t_i) \approx K$, any $u_i \sqsubset_K \mathit{term}(t_i)$ has a sub-term u' such that $u' \sqsubset_K \mathit{term}(n_{i+1})$ and $u' \neq \texttt{INH}$, by **Lemma** 5.
- As $n_{i+1}\#a_{i+1}$ is an evaluable higher-order attribute instance, we have $u' \overset{\mathcal{R}}{\Longrightarrow}^{+} u_{i+1}$ for some $u_{i+1} \sqsubset_K \mathit{term}(t_{i+1})$, by **Lemma** 6.
- Since $\mathit{type}_a(a_{i+1}) \approx K$, we have $u_{i+1} \neq \texttt{INH}$, by **Lemma** 6.
- We define $R(t_{i+1})$ to be $R(t_i)[u' \rightsquigarrow u_{i+1}]$.
- Then there exists a sub-term u_{i+1} of $R(t_{i+1})$ such that $u_{i+1} \sqsubset_K \mathit{term}(t_{i+1})$, $u_{i+1} \neq \texttt{INH}$ and $R(t_i) \overset{\mathcal{R}}{\Longrightarrow}^{+} R(t_{i+1})$.

So if the rules terminate, then all constant tree creation sequences terminate. Ordering non-terminals as described in Section 5.4 allows us to limit the number of inherited accesses in a tree creation sequence, and then use rewrite rules to model the parts of the sequence that do not use inherited attributes. Thus for a grammar whose non-terminals can be ordered as described above, and whose rules terminate, tree creation always terminates. This is because if the non-terminals can be ordered as required, then for an infinite tree creation sequence, there is an infinite constant tree creation sequence. And for an infinite constant tree creation sequence, we can construct an infinite rewrite sequence, which contradicts the assumption that the rules are terminating. We state this formally as **Theorem** II.

Theorem II: For a grammar with productions P and non-terminals NT, if

- an ordering $\succeq$ exists on NT that satisfies the conditions in Figure 5.2, and
- if the rewrite rules in $getRules(P)$ are terminating,

then there is no improper evaluation sequence for any valid tree in the grammar.

Proof: The proof is by contradiction.

- Assume an ordering $\succeq$ on NT that satisfies the conditions in Figure 5.2.
- Assume there is an improper evaluation sequence.
- By **Theorem** I, there is an infinite tree creation sequence.
- By **Lemma** 2, there is therefore an infinite constant tree creation sequence $\langle t_0, \Gamma_0 \rangle, \langle t_1, \Gamma_1, n_1\#a_1 \rangle$, where $t_0 \approx t_1 \approx t_2 \approx ...$
- Let $\mathcal{R} = getRules(P)$.
- By **Lemma** 7 we can define a function R so that for $i > 0, R(t_{i-1}) \overset{\mathcal{R}}{\Longrightarrow}^{+} R(t_i)$.
- Thus there is an infinite rewrite sequence $R(t_0) \overset{\mathcal{R}}{\Longrightarrow}^{+} R(t_1) \overset{\mathcal{R}}{\Longrightarrow}^{+} R(t_2) \overset{\mathcal{R}}{\Longrightarrow}^{+} ...$
- Thus if there is an improper evaluation sequence, either the rules in $getRules(P)$ are non-terminating, or $\succeq$ does not exist.
- Thus if the rules are terminating and $\succeq$ exists, there is no improper evaluation sequence.

5.5.3 Analysing the Rules for Termination by Ordering Productions

In this section, we describe a simple procedure that checks the generated rules for termination. The rules constructed by the analysis are simple enough that for many interesting grammars, they can be shown to be terminating via the construction of an ordering on production symbols based on how they occur in the rules. In cases where this production ordering procedure cannot show rule termination, we can use more powerful external tools such as AProVE.

Given a partial ordering $\succ$ on a set $\mathcal{F}$ of term constructors (such as production and terminal symbols), the recursive path ordering $\succ^*$ is a related ordering on $\mathcal{S}(\mathcal{F})$, the set of terms constructed using the symbols in $\mathcal{F}$. $\succ^*$ is defined as follows [38]. Say $f, g \in \mathcal{F}$ and $s_1, ..., s_m, t_1, ..., t_n \in \mathcal{S}(\mathcal{F})$. Then,

$$s = f(s_1, ..., s_m) \succ^* g(t_1, ..., t_n) = t$$

if and only if

- $f = g$ and $\{s_1, ..., s_m\} \gg^* \{t_1, ..., t_n\}$, or
- $f \succ g$ and $\{s\} \gg^* \{t_1, ..., t_n\}$, or
- $f \not\succeq g$ and either $\{s_1, ..., s_m\} \gg^* \{t\}$ or $\{s_1, ..., s_m\} = \{t\}$

where $\gg^*$ is the extension of $\succ^*$ to multisets. In other words, $\gg^*$ is a partial ordering on sets of elements in $\mathcal{S}(\mathcal{F})$ in which there may be multiple copies of each element. If S_1 and S_2 are multisets of elements in $\mathcal{S}(\mathcal{F})$, $S_1 \gg^* S_2$ if and only if we can obtain S_2 from S_1 by replacing one or more of S_1's elements with finitely many elements that are all smaller (with respect to $\succ^*$) than one of the removed elements.

The usefulness of recursive path orderings arises from the fact that they can be used to show that a given set of rewrite rules is terminating. Say $\succ$ is a partial ordering on $\mathcal{F}$. Say $\mathcal{R}$ is a set of rules on $\mathcal{S}(\mathcal{F})$. If each $r \longrightarrow s \in \mathcal{R}$ is such that $r \succ^* s$, then $\mathcal{R}$ is terminating, by Dershowitz' First Termination Theorem [38].

We can use this fact to check the rules generated by the termination analysis for termination. In this case, the term constructors are elements of the set $P \ \cup \ T \ \{\texttt{INH}\}$ and the rewrite rule terms are given by the type *RTerm*. The rules are simple enough that we can successfully generate an ordering based on the occurrence of production symbols in the rules, and prove that the rules satisfy the corresponding recursive path ordering, for many interesting grammars.

Figure 5.7 gives a sound and terminating procedure that attempts to construct an ordering on a given grammar's productions so that its rules satisfy the corresponding recursive path ordering.

An ordering $\succ$ constructed by **Procedure** C guarantees that each generated rule in *getRules*(P) satisfies its recursive path ordering $\succ^*$. We state this as **Lemma** 8.

Procedure C:

1. Construct a directed graph G^P whose vertices correspond to the grammar productions in P, and with an edge $\langle\ p,\ \ q\ \rangle$ if and only if q appears in the right-hand side of a rule with left-hand side $p(x_1, ..., x_m)$.
2. If G^P has cycles, return failure.
3. The desired ordering is defined as follows:
   - for any $p, q \in P$, $p\ \succ\ q$ if and only if there is a (possibly trivial) path in G^P from p to q
   - $p \succ c$ for all $p \in P,\ c \in T$
   - $p \succ$ `INH` for all $p \in P$

Figure 5.7: A procedure that attempts to construct an ordering on a given grammar's productions so that its rules satisfy the corresponding recursive path ordering.

Lemma 8: If an ordering $\succ$ on $P\ \cup\ T\ \{\texttt{INH}\}$ exists such that for any rule $p(\#1, ..., \#n_p) \longrightarrow r \in \mathit{getRules}(P)$ we have $p\ \succ\ q$ for every production q that appears in r, then every rule $p(\#1, ..., \#n_p)\ \longrightarrow\ r\ \in\ \mathit{getRules}(P)$ is such that $p(\#1, ..., \#n_p)\ \succ^*\ r$.

Proof: Proof is by induction on the structure of r.

Base Cases:

- r is $\#i$:

  - As any recursive path ordering is a simplification ordering [38], meaning that it holds on sub-terms, we have $p(\#1, ..., \#n_p)\ \succ^*\ \#i$.

- r is c:

  - Since by definition, $p\ \succ\ c$, the problem of showing $p(\#1, ..., \#n_p)\ \succ^*\ c$ reduces to showing $\{\ p(\#1, ..., \#n_p)\ \}\ \gg^*\ \{\ \}$, which holds trivially.

- r is `INH`:

- Since by definition, $p \succ$ `INH`, the problem of showing $p(\#1, ..., \#n_p) \succ^*$ `INH` reduces to showing $\{\ p(\#1, ..., \#n_p)\ \} \gg^* \{\ \}$, which holds trivially.

Inductive Cases:

- r is $q(r_1, ..., r_{n_q})$:

- Since the proof by induction is on the structure of the right-hand sides of the rules, the inductive assumption in this case is that $p(\#1, ..., \#n_p) \succ^* r_i$ for $1 \leq i \leq n_q$.
- Since we have $p \succ q$, the problem of showing $p(\#1, ..., \#n_p) \succ^* q(r_1, ..., r_{n_q})$ reduces to showing $\{\ p(\#1, ..., \#n_p)\ \} \gg^* \{\ r_1, ..., r_{n_q}\ \}$.
- This holds since the inductive assumption allows us to replace $p(\#1, ..., \#n_p)$ in $\{\ p(\#1, ..., \#n_p)\ \}$ with $r_1, ..., r_{n_q}$ to obtain $\{\ r_1, ..., r_{n_q}\ \}$.

The rules generated for grammar G_1 defined in Appendix B.1.2 can be shown to be terminating using **Procedure** C. In this case the productions are ordered as follows:

$$\{\texttt{doWhile} \succ \texttt{consStmt},\ \texttt{doWhile} \succ \texttt{while}, \texttt{ifThen} \succ \texttt{ifThenElse}, \texttt{ifThen} \succ \texttt{emptyStmt}\}$$

Similarly, the rules generated for grammar G_2 defined in Appendix B.2.2 can be shown to be terminating using a production ordering.

5.5.4 A Modular Version of the Production Ordering Procedure

We can specify a condition on extensions that guarantees that a valid production ordering exists for any combination of extensions that satisfy it. We also define a corresponding modular monolithic version of the production ordering procedure described in the previous section. Section 2.2.3 gives a short note on composing higher-order attribute grammars. We write $p \overset{G}{\succ} q$ if $p \succ q$ under the grammar G.

An extension passes the modular production ordering if it introduces no new ordering relation on host language productions. In other words, for a host language G_H with productions P_H, an extended grammar G_{H+E} passes the modular condition if and only if, for any two productions $p, q \in P_H$,

$$p \overset{G_{H+E}}{\succ} q \Longrightarrow p \overset{G_H}{\succ} q$$

We state this formally as **Lemma** 9.

Lemma 9:

- Say the host language grammar $G_H =$
  $\langle\ \langle\ NT_H,\ T_H,\ P_H,\ S\ \rangle,\ PT,\ PV,\ F,\ A_S,\ A_I,\ L_H,\ \mathit{defs}_H\ \rangle$
  has an ordering $\overset{G_H}{\succ}$ on P_H, such that for any rule $p(\#1, ..., \#n_p) \longrightarrow r \in$ $\mathit{getRules}(P_H)$ we have $p\ \succ\ q$ for every production q that appears in r.

- Say there are n extended grammars where for $1 \leq i \leq n$, each extended grammar $G_{H+E_i} =$
  $\langle\ \langle\ NT_H\ \cup\ NT_{E_i},\ T_H\ \cup\ T_{E_i},\ P_H\ \cup\ P_{E_i},\ S\ \rangle,\ PT,\ PV,\ F,$
  $A_{SH}\ \cup\ A_{SE_i},\ A_{IH}\ \cup\ A_{IE_i},\ L_H\ \cup\ L_{E_i},\ \mathit{defs}_H\ \cup\ \mathit{defs}_{E_i}\ \rangle$
  has an ordering $\overset{G_{H+E_i}}{\succ}$ on $P_H \cup P_{E_i}$, such that for any rule $p(\#1, ..., \#n_p) \longrightarrow$ $r \in \mathit{getRules}(P_H\ \cup\ P_{E_i})$ we have $p\ \succ\ q$ for every production q that appears in r, where for all $p, q \in P_H$
  $p\ \overset{G_{H+E_i}}{\succ}\ q \Longrightarrow p\ \overset{G_H}{\succ}\ q.$

then the rules generated when the host language is composed with these extensions (given by $\mathit{getRules}(P_H\ \cup\ P_{E_1}\ \cup\ P_{E_2}\ ...\ \cup\ P_{E_n})$) are terminating.

Proof:

- Let G^P_H be the production graph of the host language H.
- Let $G^P_{H+E_i}$ be the production graph of the extended grammar $H + E_i$.
- Let $G^P_{H+E_1+...+E_n}$ be the production graph of the host language composed with $E_1, ..., E_n$.
- $G^P_{H+E_1+...+E_n}$ can be obtained by taking the union of the production graphs for each extension. Its vertices are given by $P_H\ \cup\ P_{E_1}\ \cup ... \cup\ P_{E_n}$.
  $G^P_{H+E_1+...+E_n}$ has an edge $\langle p, q\rangle$ if and only if
  - $p, q \in P_H$, $\langle p, q\rangle$ is an edge in G^P_H, or
  - $p, q \in P_H\ \cup\ P_{E_i}$, $\langle p, q\rangle$ is an edge in $G^P_{H+E_i}$ for some extended grammar $H + E_i$.

- We need to show that $G^P_{H+E_1+...+E_n}$ has no cycles.
- Assume there is a cycle in $G^P_{H+E_1+...+E_n}$.
- The cycle cannot consist solely of productions belonging to a single extension (since it passed the monolithic analysis), and there are no edges between productions of different extensions.
- Thus the cycle consists of sections bounded by host language productions such that all productions in between belong to a single extended grammar.
- Consider such a section $h_1, v_1, ..., v_n, h_2$ where $h_1, h_2 \in P_H$ and $v_1, ..., v_n \in P_{H+E_i}$, for some extended grammar $H + E_i$.
- Since the corresponding path is present in G_{H+E_i}, we have $h_1 \overset{G_{H+E_i}}{\succ} h_2$ and therefore $h_1 \overset{G_H}{\succ} h_2$ by the modularity condition.
- We can repeat this for each of the sections in the cycle in $G^P_{H+E_1+...+E_n}$, so that we have a cycle in G^P_H, which is a contradiction (as we assume the host language passed the monolithic analysis).
- Therefore $G^P_{H+E_1+...+E_n}$ has no cycles and the composed grammar passes the monolithic analysis.

To check whether an extension E satisfies the modular ordering condition, we compare G^P_H and G^P_{H+E}, the host language H's and the extended grammar's production graphs, using the procedure in Figure 5.8. For each pair of host language productions, we verify there is a path between them in G^P_{H+E} only if there is one in G^P_H. Note that this same principle can be used on extensions that depend on other extensions; in this case the "host language" would be the actual host language composed with the other required extensions.

5.6 Discussion

In this chapter, we presented a static analysis on higher-order attribute grammars that detects non-terminating tree creation during attribute evaluation. The analysis, in conjunction with the standard completeness and circularity tests for higher-order attribute grammars, provides extension writers and programmers with a guarantee that the compilers generated by the Silver tool will not fail to terminate on account of improper

Procedure D:

1. Construct G_H^P and G_{H+E}^P, the host language's and the extended grammar's production graphs, as described in **Procedure** C.
2. For each p in P_H
   (a) For each q in P_H
      i. If there is a path from p to q in G_{H+E}^P, and if there is *no* path from p to q in G_H^P, then return failure.
3. Return success.

Figure 5.8: A modular analysis that checks extension grammars for restrictions that ensure that **Procedure** C succeeds on any combination of this extension with other well-behaved extensions.

attribute evaluation.

The problem of showing termination of higher-order evaluation during attribution is undecidable, so the best we can hope for is a conservative analysis. Our analysis incorporates domain-specific knowledge of attribute grammars as well as the information we have gained from implementing attribute grammar-based specifications for real-world programming languages to generate rules that are simpler and easier to analyse than the programs that implement the corresponding attribute evaluator. The rules are conservative in how they model tree creation since they derive more terms than are actually generated during higher-order evaluation. But they are useful and capable of showing termination of grammars such as ABLEJ in a reasonable amount of time.

The analysis is sound; we have shown formally that if the specified non-terminal ordering exists, and the generated rules are terminating, then there are no infinite tree creation sequences and therefore attribute evaluation terminates. The termination analysis is simple enough to be easily proven correct and efficiently executed, but is also useful enough to show termination for a large class of useful and interesting grammars.

Modular versions of the restrictions enforced by the monolithic termination analysis have also been defined. These, if satisfied by an extension specification, guarantee that any combination of that extension with other "well-behaved" extensions passes the monolithic test. This is an important step toward our goal of a library model of language extensibility in which extension specifications can be automatically composed by application programmers with no compiler expertise.

5.6.1 Running the Analysis on Sample Grammars

In this section, we look at the results of running the monolithic and modular versions of the termination analysis on the sample grammars listed in Section 5.2.1, and the ableJ host language and extension specifications. The sample grammars are the following:

- G_0 is the the grammar (shown in Figure 5.1) defined by Vogt *et al.* [17].
- G_1 is described in Appendix B.1 and specifies a small imperative language, using local attributes to simulate forwarding.
- G_2 is described in Appendix B.2 and specifies a small imperative language with multiple levels of inherited attribute occurrences.
- G_3 is described in Appendix B.3 and extends G_0 to compute a simple "host language translation".

The ableJ specification [19] is described in Chapter 4. The ableJ extensions analysed are the following:

- The complex number type extension described in Section 4.1.2.2.
- An expression block extension that uses anonymous classes to associate expressions with statement blocks.
- The enhanced `for` loop extension described in Section 4.1.2.1.
- The Pizza-style algebraic data-type extension described in Section 4.6.1.
- An extension for programming in the domain of computational geometry [30].
- The SQL extension described in Section 4.6.2; this module requires the enhanced `for` loop module.
- An extension that implements condition tables from modeling languages [32, 33]); this module requires the expression block module.

Table 5.1 shows the results of running the monolithic production and non-terminal ordering analyses on the sample grammars listed above. The non-terminal ordering procedure succeeded on every grammar while the generated rules were terminating for

Grammar	NT Ordering exists?	Production ordering exists?	AProVE shows termination?	Termination shown?
G0	YES	NO	NO	NO
G1	YES	YES	YES	YES
G2	YES	YES	YES	YES
G3	YES	NO	NO	NO

Table 5.1: Running the monolithic production and non-terminal ordering procedures on sample grammars.

G_1 and G_2. G_0 is the example developed by Vogt [17] to illustrate non-terminating tree creation. G_3 simulates forwarding and defines a simple "host language translation". As described in Appendix B.3, most practical evaluators would terminate when evaluating this grammar as these "translation attribute instances" would be evaluated only once each. However, our evaluation model evaluates all attribute instances and is non-terminating for this grammar, and our analysis correspondingly fails.

A simple example of the limits of our analysis is given by the grammar G_4, a specification of a let-expression language that we used in our descriptions of attribute grammars (in Chapter 2) and Silver (in Chapter 3). This grammar is incomplete and therefore does not satisfy the prerequisites for being analysed by our termination analysis. But it demonstrates a situation where the analysis fails for a terminating grammar. Since `Expr` has an inherited attribute `env` of non-terminal type `Env`, and also appears on the right-hand side of a production `consEnv` with `Env` on its left-hand side, the desired non-terminal ordering cannot be constructed. But since the expressions which appear in the environment are integer constants whose environments would be empty or left unevaluated, this entanglement between the `Expr` and `Env` non-terminals does not actually result in non-terminating tree creation. Our analysis is too coarse to take this information into account, and so is too imprecise to show termination for this grammar.

Tables 5.2 and 5.3 show the results of running ableJ through the analysis. We made some minor modifications to some of the grammars to get them to satisfy the restrictions that define RHOAG and to get them to pass the analysis. Note that since our restricted class of grammars does not include functions, the analysis does not examine parts of the ableJ grammar that define Silver functions for purposes such as type conversion. Similarly, the SQL extension uses reference attributes which our analysis does not handle.

Grammar	NT ordering exists?	Production ordering exists?	AProVE shows termination?	Term. shown?
ABLEJ	YES	YES	YES	YES
ABLEJ+ complex	YES	YES	YES	YES
ABLEJ+ exprblock	YES	YES	YES	YES
ABLEJ+ foreach	YES	YES	YES	YES
ABLEJ+ pizza	NO	YES	YES	NO
ABLEJ+ CG	YES	NO	NO	YES
ABLEJ+ foreach + SQL	YES	YES	YES	YES
ABLEJ+ exprblock + tables	NO	YES	YES	NO

Table 5.2: Running the monolithic production and non-terminal ordering procedures on ABLEJ grammars.

And so the results shown are for those parts of the extension that do not use these Silver features. Future work will explore handling these non-standard features.

Table 5.2 shows the results of running the monolithic production and non-terminal ordering analyses on the ABLEJ host language and extension grammars. The non-terminal ordering procedure succeeded with every extension except the Pizza and condition-table extensions. The procedure fails on these two extensions because of the way they decorate non-terminals such as expressions with inherited attributes of the same type to generate host language translated code, such as the case parameters within pattern-matching expressions in the Pizza extension.

Table 5.3 shows the results of running the modular production and non-terminal ordering analyses on the ABLEJ extension grammars. All the extensions that passed the monolithic non-terminal ordering procedure passed the modular version. Some of them also passed the modular production ordering procedure; others failed because they added new production ordering relations in their local attribute definitions, for various reasons.

We observe that the production ordering procedure had the same results as AProVE when analysing the generated rules for termination. Thus, in their current shape, it is not necessary to leverage powerful tools such as AProVE in the analysis. But future work could exploit them to analyse a larger class of grammars.

Grammar	NT ordering is modular?	Prod. ordering is modular?	Modular term. shown?
ABLEJ+ complex	YES	NO	NO
ABLEJ+ exprblock	YES	YES	YES
ABLEJ+ foreach	YES	YES	YES
ABLEJ+ pizza	N.A.	NO	N.A.
ABLEJ+ CG	YES	N.A.	N.A.
ABLEJ+ foreach + SQL	YES	NO	NO
ABLEJ+ exprblock + tables	N.A.	YES	N.A.

Table 5.3: Running the modular production and non-terminal ordering procedures on ABLEJ grammars. The modular analyses are applicable only to those extensions that pass their monolithic versions.

To conclude, the analysis succeeded on many of the extensions, but failed on others, for various reasons. These extensions were written before the analysis was developed and would likely be written differently now that we have the analysis, as well as the informal invariants referenced in Chapter 4. But the fact that the termination analysis successfully handles grammars as complex as ABLEJ demonstrates its usefulness and general applicability. It is an important first step toward our goal of modular composable extension specifications.

Chapter 6

Related Work and Conclusion

This chapter concludes the dissertation. Section 6.1 gives an overview of related work. Section 6.2 looks at opportunities for future work. Section 6.3 concludes with final thoughts.

6.1 Related Work

Early approaches to language extensibility such as embedded DSLs [39], and traditional syntactic, hygienic and programmable macro systems [7, 8, 40], provided limited facilities for semantic analysis of new constructs, and little domain-specific feedback to the programmer. Meta-object protocol systems [41] and modern macros [42] offer more opportunities for optimization. Further, traditional approaches to extensibility are restricted in their scope. They either add language features from specific domains, or handle only certain aspects of language extensibility such as adding new syntax or source-level optimizing transformations. They do not deal with the complications involved when composing multiple extensions, such as the dependencies between them.

The extensible Java tool JavaBorg [9] allows the addition of new concrete syntax for objects. It is based on the MetaBorg embedding tool [43] that uses scanner-less GLR parsers. It performs generative, optimizing transformations via destructive rewrites. The underlying Stratego/XT rewriting system [44] allows program transformation specifications via conditional rewrite rules on abstract syntax trees. It also allows for meta-strategies that programmatically specify how rules are dynamically constructed and applied. These meta-strategies can be composed and bundled into libraries.

Attribute grammar based-language tools such as Eli [45] (which like Silver, is functional), LRC [46] and JastAdd [10] implement domain-specific translations in different ways. JastAdd is an object-oriented extensible compiler tool that has been used to build a Java 1.5 extensible compiler. In JastAdd, extensions are specified as rewritable reference attribute grammars (RAG), where reference attribute values are pointers to other (decorated) nodes [27]. Like Silver, JastAdd specifies extension semantics implicitly via existing host constructs, but it does this by performing destructive rewrites on decorated syntax trees. It does not allow explicit attribute definitions on the non-rewritten extension constructs. JastAdd's destructive and more general rewriting framework can perform more invasive optimizing tree transformations than the more specialized notion of forwarding allows directly. But this comes with a more difficult problem of ensuring termination of the tree rewriting process. The semantics of attribute evaluation interleaved with destructive tree rewriting is harder to intuit. Thus JastAdd includes no static analysis for rewrite termination and performs only run-time checks.

This is in contrast with forwarding in Silver, which preserves both extension and host trees during attribution. Forwarding increases modularity by allowing for reuse of some host semantic specifications while specifying new domain-specific analyses on extension constructs. Intentional Programming [47] used a notion similar to forwarding in a non-attribute grammar setting. The functional nature of attribute evaluation in Silver and the semantics of forwarding allow for more analysis of termination than an object-oriented framework such as JastAdd. The price of this increased reliability is that destructive rewrites can only be achieved via more complicated specifications that use higher-order attribute definitions to construct the desired trees.

Thus the main differences between Silver and other attribute-grammar based extensible language tools are Silver's notion of forwarding which aids in the modular writing and automatic composition of extensions without glue code, its high-level language features such as pattern-matching and polymorphic lists, and most significantly, its incorporation of syntactic and semantic analyses to increase the reliability of generated compilers. The extensions specifiable in Silver are often beyond the scope of other frameworks or approaches (such as SugarJ [48] or Kiama [49]), both in terms of their functionality and their ability to be composed. And unlike systems like Polyglot [50] which require the specification of the order of code transformations, Silver aims to shield users from detailed implementation-level knowledge when composing extensions.

Knuth presented a circularity analysis in his original work on canonical attribute grammars [16]. This analysis has been extended to handle new features as they are added to the attribute grammar formalism. Vogt *et al.* extended it to handle higher-order attribute grammars [17]. With forwarding, the circularity analysis was combined with the completeness analysis to incorporate global attribute dependency information in the latter [23, 51]. For reference and remote attributes [27, 24], the problem is undecidable [24] and therefore no precise circularity analysis is possible.

As we mentioned in Section 5.1.1, Vogt *et al.* listed conditions for higher-order attribute grammars to be well-defined (i.e, for attribute evaluation to terminate normally). One of these imposes a sufficient condition for tree creation to be terminating, viz, "on every path in every structure tree, a particular non-terminal attribute occurs at most once"[17, Lemma 3.2, page 142], which in our model translates approximately to requiring that each higher-order attribute occurs at most once in a particular tree creation path. This is clearly similar to our non-terminal ordering analysis that attempts to limit the presence of higher-order inherited attributes within tree creation sequences. But our use of partial orders and rewrite rules generated from the definitions of local and synthesized higher-order attribute definitions, is not as restricting, and results in a more precise termination analysis.

There is much related work in the field of analysing functional and imperative programs for termination. An example is the work by Lee, Jones and Ben-Amram[52] on applying the size-change principle to first order functional programs (extended by Sereni and Jones [53] to higher-order functional programs), where each function call's parameter must decrease over a domain with a least value. These efforts are targeted at showing termination of programs written in general purpose programming languages. It is possible to use these techniques on functional program translations [54] of Silver attribute grammars (such as the Haskell code formerly generated by the Silver compiler). But, as we have stressed, using domain knowledge from the attribute grammar formalism results in an analysis that is simpler and usable on large grammars.

6.2 Future Work

In this section we look at possible future work to extend this research.

Handling More of Silver's Specification Language

The restricted class of higher-order attribute grammars handled by our termination analysis does not cover all the features present in Silver's specification language. Some of these features (such as aspect productions and production attributes) are not relevant to the process of tree creation examined by the analysis. Others (such as forwarding, collection attributes, function declarations and pattern-matching) can be automatically converted to equivalent constructs in the restricted class via a simple and terminating procedure. Future work could therefore extend the analysis to handle these and any remaining features in Silver such as case expressions and reference attributes.

Handling Functions with Inductive Definitions

Our analysis does not attempt to show termination for attribute grammars that implement inductively defined functions. Thus the analysis cannot show termination of many grammars such as the Strong Attribute Grammars (SAG) described by Saraiva [55]. The simple production ordering procedure described in Section 5.5.3 would not succeed on the rules that would be generated for inductive functions. However more powerful analyses and tools, such as AProVE, do handle many inductive function definitions, such as the factorial function, and list look-ups. Future work could therefore extend the analysis to handle some inductive function definitions. While the resulting analysis might not be as general as AProVE, it might be possible to develop a modular version of it.

6.3 Concluding Thoughts

Libraries are a useful means of introducing abstractions into existing languages, primarily because the programmer can import those libraries that provide solutions to his specific problems, freely and without detailed information on their implementation. We believe programmers must be able to compose language extensions in a similar way for them to have real-world impact. Our experience tells us that higher-order attribute grammar frameworks such as Silver are an appropriate tool to the goal of modular and composable semantic and syntactic language specifications. We have shown that implementations can be designed for general-purpose languages so that they can be augmented with useful general-purpose and domain-specific features, specified as modular and composable extensions to a host specification of the language. Domain-specific features can be packaged into self-contained extension specifications that define new language constructs by specifying their syntax and by specifying domain-specific semantic analyses that perform compile-time source-code optimizations, and perform error-checking that provides useful domain-appropriate feedback to the programmer. Using a declarative formalism such as higher-order attribute grammars makes it easier to develop static specification analyses that ensure that the generated compiler terminates.

In this dissertation, we presented a host language implementation (ableJ) for a non-trivial imperative language (Java 1.4). We looked at how host language specifications can be designed for interesting extensions, with functionality beyond simple macros, and incorporating invariants that ensure composability. We also exploited domain-specific features present within the attribute grammar formalism to develop an analysis for showing termination of generated compilers. The contributions of this dissertation both relate to composability of extensions with respect to semantics. The first relates to the actual writing of specifications, the other to showing the termination of generated compilers.

We conclude with the notion that it is worthwhile to explore the broad area of language extensibility in a systematic fashion. Existing approaches to language extensibility, whether based on using general-purpose programming languages, or domain-specific languages are *ad hoc*. Work in this field would benefit from stating formal criteria by which to measure the quality of any approach or tool. Some criteria for success while developing tools in this field are necessarily informal. Examples include the experiences of application programmers, the domains that might be ripe for abstraction and the ease of use and learning curve associated with any extensible language framework. Other criteria are less vague. These include checking a grammar to see if a parser can be generated, or if higher-order attribute evaluation terminates, or whether modular versions of these analyses exist. We believe that the work described in this dissertation, both with respect to developing principles and best practices while designing host languages and extensions, and with respect to developing static analyses for detecting termination of higher-order attribute evaluation, are useful first steps toward a systematic approach to programming language extensibility.

References

[1] Philip Wadler. The expression problem. 1998. Note to the Java Genericity mailing list, 12 November 1998.

[2] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[3] C. J. Date and Hugh Darwen. *A Guide to the SQL Standard, 4th Edition*. Addison-Wesley, 1997.

[4] Donald E. Thomas and Philip R. Moorby. *The Verilog® hardware description language, 5th Edition*. Springer, 2002.

[5] Amos Gilat. *MATLAB: An Introduction with Applications, 4th Edition*. John Wiley & Sons, 2010.

[6] Arch D. Robison. Impact of economics on compiler optimization. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, pages 1–10. ACM Press, 2001.

[7] T. E. Cheatham, Jr. The introduction of definitional facilities into higher level programming languages. In *Proceedings of the Fall Joint Computer Conference, AFIPS*, pages 623–637, 1966.

[8] B. M. Leavenworth. Syntax macros and extended translation. *Communications of the ACM*, 9(11):790–793, 1966.

[9] Karina Olmos and Eelco Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In *Proceedings*

of the 14th International Conference on Compiler Construction, volume 3443 of *LNCS*, pages 204–220, 2005.

[10] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *Proceedings of the Conference on Object-oriented Programming Systems and Applications*, pages 1–18. ACM, 2007.

[11] Eric Van Wyk, Derek Bodin, Lijesh Krishnan, and Jimin Gao. Silver: An extensible attribute grammar system. *Electronic Notes in Theoretical Computer Science*, 203(2):103–116, 2008.

[12] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[13] Eric Van Wyk and August Schwerdfeger. Context-aware scanning for parsing extensible languages. In *International Conference on Generative Programming and Component Engineering*. ACM Press, October 2007.

[14] August Schwerdfeger and Eric Van Wyk. Verifiable composition of deterministic grammars. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 2009.

[15] August Schwerdfeger. *Context-Aware Scanning and Determinism-Preserving Grammar Composition, in Theory and Practice*. PhD thesis, University of Minnesota, Minneapolis, MN, USA, 2010.

[16] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Corrections in **5**(1971):95–96.

[17] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher-order attribute grammars. In *ACM Conference on Programming Language Design and Implementation*, pages 131–145, 1990.

[18] Ted Kaminski and Eric Van Wyk. Modular well-definedness analysis for attribute grammars. In *Proceedings of the 5th International Conference on Software Language Engineering*, 2012.

[19] Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. Attribute grammar-based language extensions for Java. In *European Conference on Object Oriented Programming*, volume 4609 of *LNCS*, pages 575–599. Springer-Verlag, July 2007.

[20] Lijesh Krishnan and Eric Van Wyk. Termination analysis for higher-order attribute grammars. In *Proceedings of the 5th International Conference on Software Language Engineering*, 2012.

[21] Eric Van Wyk and Lijesh Krishnan. Using verified data-flow analysis-based optimizations in attribute grammars. *Electronic Notes in Theoretical Computer Science*, 176(3):109–122, 2007.

[22] S. Doaitse Swierstra and Harald Vogt. Higher-order attribute grammars. In *International Summer School on Attribute Grammars Applications and Systems*, volume 545 of *LNCS*, pages 256–296. Springer-Verlag, 1991.

[23] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proceedings of the 11th International Conference on Compiler Construction*, volume 2304 of *LNCS*, pages 128–142, 2002.

[24] John Tang Boyland. Remote attribute grammars. *Journal of the ACM*, 52(4):627–687, 2005.

[25] Jimin Gao. *An Extensible Modeling Language Framework via Attribute Grammars.* PhD thesis, University of Minnesota, Minneapolis, MN, USA, 2008.

[26] Ted Kaminski and Eric Van Wyk. Integrating attribute grammar and functional programming language features. In *Proceedings of the 4th International Conference on Software Language Engineering*, volume 6940 of *LNCS*, pages 263–282. Springer-Verlag, July 2011.

[27] Görel Hedin. Reference attribute grammars. *Informatica*, 24(3):301–317, 2000.

[28] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification, 2nd Edition.* Prentice Hall, 2000.

[29] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification, 3rd Edition*. Addison-Wesley Professional, 2005.

[30] Eric Van Wyk and Eric Johnson. Composable language extensions for computational geometry: A case study. In *Proceedings of the 40th Hawai'i International Conference on System Sciences*, 2007.

[31] Eric Van Wyk and Yogesh Mali. Adding dimension analysis to Java as a composable language extension. In *Post Proceedings of Generative and Transformational Techniques in Software Engineering*, volume 5235 of *LNCS*, pages 442–456. Springer-Verlag, 2008.

[32] Jimin Gao, Mats Heimdahl, and Eric Van Wyk. Flexible and extensible notations for modeling languages. In *Fundamental Approaches to Software Engineering*, volume 4422 of *LNCS*, pages 102–116. Springer-Verlag, March 2007.

[33] Jimin Gao, Michael Whalen, and Eric Van Wyk. Extending Lustre with timeout automata. *Electronic Notes in Theoretical Computer Science*, 203(4):111–124, 2008.

[34] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of Symposium on Principles of Programming Languages*, pages 146–159. ACM Press, 1997.

[35] Eric Van Wyk, Derek Bodin, and Paul Huntington. Adding syntax and static analysis to libraries via extensible compilers and language extensions. In *Proceedings of Library-Centric Software Design*, 2006.

[36] Harald Vogt. *Higher order attribute grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, 1989.

[37] J urgen Giesl, René Thiemann, Peter Schneider-kamp, and Stephan Falke. Aprove: A system for proving termination. In *Extended Abstracts of the 6th International Workshop on Termination*, pages 68–70, 2003.

[38] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[39] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es), 1996.

[40] Daniel Weise and Roger Crew. Programmable syntax macros. *ACM SIGPLAN Notices*, 28(6), 1993.

[41] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the MetaObject Protocol*. MIT Press, 1991.

[42] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Proceedings of the 5th International Conference on Software Reuse*. IEEE, 2–5 1998.

[43] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, pages 365–383, 2004.

[44] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 216–238. Springer-Verlag, June 2004.

[45] Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35:121–131, 1992.

[46] Matthijs F. Kuiper and João Saraiva. LRC — A generator for incremental language-oriented tools. In *Proceedings of the 7th International Conference on Compiler Construction*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, 1998.

[47] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. *ACM SIGPLAN Notices*, 41(10):451–464, 2006.

[48] Sebastian Erdweg, Tillmann Rendel, Christian K astner, and Klaus Ostermann. SugarJ: Library-based syntactic language extensibility. In *Proceedings of the Conference on Object Oriented Programming, Languages, and Systems*. ACM, 2011.

[49] Anthony M. Sloane. Lightweight language processing in Kiama. In *Proceedings of the 3rd Summer School on Generative and Transformational Techniques in Software Engineering*, pages 408–425. Springer-Verlag, 2011.

[50] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myer. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 138–152. Springer-Verlag, 2003.

[51] Kevin Backhouse. A functional semantics of attribute grammars. In *Proceedings of the 7th Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 142–157. Springer-Verlag, 2002.

[52] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 81–92. ACM Press, 2001.

[53] Damien Sereni and Neil D. Jones. Termination analysis of higher-order functional programs. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, volume 3780 of *LNCS*, pages 281–297. Springer-Verlag, 2005.

[54] T. Johnsson. Attribute grammars as a functional programming paradigm. In *Proceedings of Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 154–173. Springer-Verlag, 1987.

[55] João Saraiva and S. Doaitse Swierstra. Generating spreadsheet-like tools from strong attribute grammars. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of *LNCS*, pages 307–323. Springer Berlin / Heidelberg, 2003.

[56] Bernhard Steffen. Data flow analysis as model checking. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Science*, volume 526 of *LNCS*, pages 346–364. Springer-Verlag, September 1991.

[57] David A. Schmidt and Bernhard Steffen. Data-flow analysis as model checking of abstract interpretations. In G. G. Levi, editor, *Proceedings of the 5th Static Analysis Symposium*, volume 1503 of *LNCS*. Springer-Verlag, 1998.

[58] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244–263, 1986.

[59] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model verifier. In *Proceedings of Computer Aided Verification*, pages 495–499, 1999.

Appendix A

Extending Silver for Data-Flow Analysis

This appendix presents a second example of a useful extension to a non-trivial programming language (in addition to the ableJ grammar described in Chapter 4). Here, the extension is written to the Silver specification language (described in Chapter 3). It provides extension writers with constructs to specify and execute compile-time data-flow analyses.

A drawback in using attribute grammar-based frameworks such as Silver to specify language semantics is the difficulty in specifying non-syntax directed semantic analyses, such as data-flow analyses, which are more naturally specified on control-flow graphs. Temporal logics such as the Computational Tree Logic (CTL) provide a declarative framework with which to specify and check data-flow conditions, by model-checking them on program control-flow graphs.

In this appendix we describe an extension to the Silver specification language that for an imperative language specification:

- provides constructs to specify how the control-flow graph is to be constructed and labeled with attribute values, and generates the NuSMV version of the graph,
- extends the syntax of attribute definitions to include CTL formulas, and
- returns the results from the NuSMV model-checker of checking CTL formulas on the control-flow graph, for use within attribute definitions.

Higher-order tree-valued attributes can be used to construct the optimized version of the program, based on the results of executing the data-flow analysis.

Appendix Outline This appendix is structured as follows:

- In Section A.1, we look at specifying the data-flow side-conditions of program transformations as temporal logic formulas which can be model-checked on the program control-flow graph.

```
  {
1   int x, y, z;
2   x = 0;
3   y = 1;
4   z = 3;
5   if (x == 1)
6       y = 5;
7   else y = 2;
8   while (y == 3)
9       x = x + 1;
  }
```

```
{
    int x, y, z;
    x = 0;
    if (x == 1)
        y = 5;
    else y = 2;
    while (y == 3)
        x = x + 1;
}
```

Figure A.1: Performing dead-code elimination on a *C–* program.

- In Section A.2, we look at constructing the nodes and labels of a program's control-flow graph from its abstract syntax tree within an attribute grammar framework.
- In Section A.3, we describe a Silver extension for specifying and executing data-flow conditions during attribute evaluation.
- Section A.4 concludes.

A.1 Automatic Annotation of Parse Trees with Data-Flow Conditions

Dead-code elimination of unused assignment statements is a well-known example of a program transformation with a data-flow side-condition. The transformation performs data-flow analysis on the program to find assignments whose values are not used on any execution path following the assignments. These assignments are removed to construct the optimized version of the program. Figure A.1 shows the results of performing this transformation on a program in the language *C–*. *C–* is a subset of C that retains only basic statements, declarations and expressions. In the original program shown in the left column, the assignments to the identifier `y` in line 3 and to the identifier `z` in line 4, are dead as data-flow analysis shows that their assignments are not used on any possible execution path. These assignments are therefore removed during dead-code elimination.

Ideally, extension writers could specify data-flow analyses declaratively, and use their results in the same framework as, and similarly to, syntax-directed analyses. There are, in fact, high-level formalisms for specifying and executing data-flow analysis conditions, such as temporal logic formulas [56, 57]. The data-flow side-conditions of certain program transformations can be specified as formulas in the Computational Tree Logic (CTL) [58]. These formulas can be model-checked on program control-flow graphs.

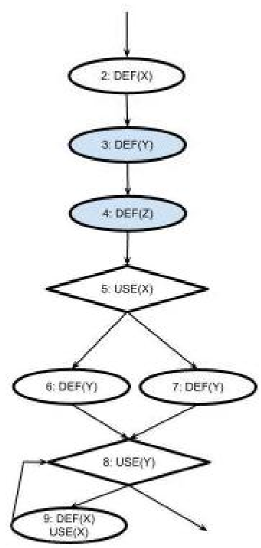


Figure A.2: The control-flow graph of the program shown in the left column of Figure A.1.

For example, the following optimization performs the dead-code elimination transformation described above, using a side-condition written in CTL.

$$\texttt{assign} : \texttt{var} := \texttt{expr} \Longrightarrow \texttt{skip}$$

$$\textbf{if } \texttt{assign} \models \mathbf{AX}\ (\mathbf{A}[\ \neg use(\texttt{var})\ \mathbf{U}\ (def(\texttt{var})\ \wedge\ \neg use(\texttt{var}))\ ]\ \vee\ \mathbf{AG}\ \neg use(\texttt{var}))$$

The transformation specifies that if a control-flow graph node `assign` (i.e., an assignment node) satisfies the given CTL formula, then the node's assignment statement is replaced with a `skip` statement. The CTL formula is true on the node `assign` if and only if the assignment to the identifier `var` is dead, i.e., if its value is not used on any execution path from the node. The meanings of CTL temporal operators are shown in Table A.1. The formula states that on **A**ll ne**X**t states from the current assignment to `var`,

- either, on **A**ll paths, `var`'s lexeme is not used **U**ntil it is redefined without using the old value,
- or, on **A**ll paths, `var`'s lexeme is never (**G**lobally) used.

Consider a CTL formula f that is checked on a state n in a model m. The temporal operators of the Computational Tree Logic are defined as follows:

- $AX\ f$: is true on n if f is true on each of n's successor states.
- $EX\ f$: is true on n if f is true on at least one of n's successor states.
- $A[\ f\ U\ g\ ]$: is true on n if on all paths from n, f holds on each state until a state on which g holds is reached.
- $E[\ f\ U\ g\ ]$: is true on n if on at least one path from n, f holds on each state until a state on which g holds is reached.
- $AG\ f$: is true on n if f holds on all states reachable from n.
- $EG\ f$: is true on n if f holds on all states on at least one path from n.

Table A.1: The temporal operators of the Computational Tree Logic.

A.2 Specifying the Control-Flow Graph in an Attribute Grammar

In this section, we look at how a program's control-flow graph can be constructed on its syntax tree. Figure A.3 shows the syntax tree of the program shown in the left column of Figure A.1. Figure A.4 shows how the program's control-flow graph's nodes correspond to some of the nodes of its syntax tree. Only a subset of syntax tree nodes, namely those dealing with control and data flow, are represented in the control-flow graph with corresponding nodes. Only these nodes have attribute instance values relevant to the data-flow analysis that will be performed on the model. Only their attribute instance values are therefore used to generate the labels of the nodes in the graph. Dead-code elimination requires the program's assignment, skip and conditional expression syntax nodes to have corresponding control-flow graph nodes. Declarations, types and similar non-flow related syntax nodes are not represented in the control-flow graph.

Control-flow graph nodes are labeled with the evaluated values of some of the attribute instances on their corresponding syntax tree nodes. The selection of label attributes is determined by the nature of the desired data-flow analysis and the information required from each node. Checking for dead code requires information about which identifiers are defined on each node, and which identifiers are referenced. In this example, two attributes are declared and defined via aspect productions for these purposes, as shown in Figure A.5. The `String`-valued synthesized attribute `def` is used for the former. It is set to the variable lexeme on assignment nodes, and to the empty string on all other nodes. The attribute `uses` is a `String` list and stores the names of identifiers referenced on its node. In Section A.3.1, we show how these attributes are used to generate node

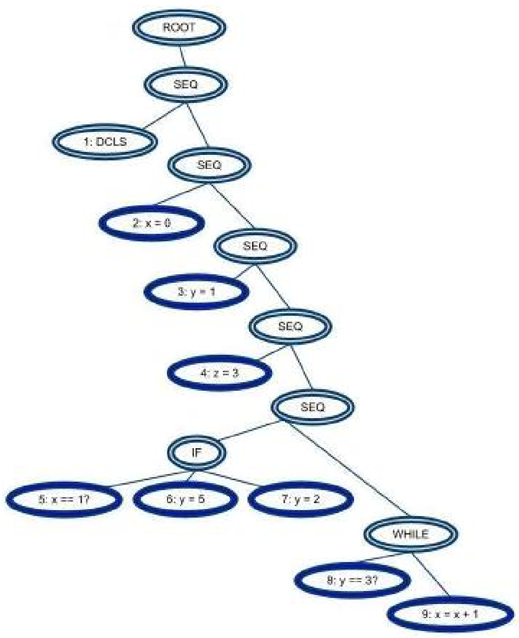


Figure A.3: The syntax tree of the program shown in the left column of Figure A.1.

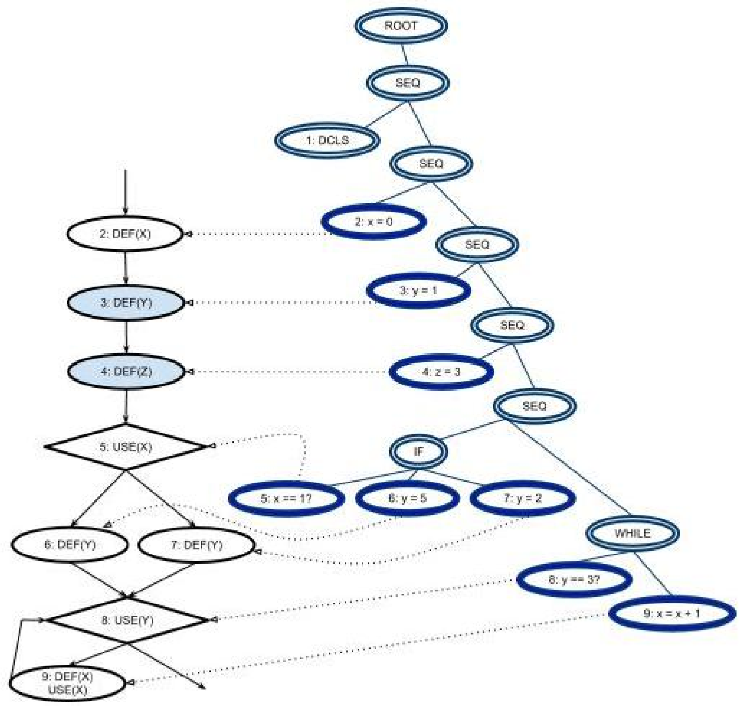


Figure A.4: Defining the control-flow graph on the syntax tree of the program shown in the left column of Figure A.1.

labels for the program's control-flow graph.

A.3 A Silver Extension for Specifying and Executing Data-Flow Conditions

There are obvious benefits to integrating these high-level representations of data-flow conditions into the Silver specification language. This would allow declaratively specified data-flow properties to be model-checked on program control-flow graphs for use in attribute evaluation, say for the construction of optimized source code. The extension described in this appendix adds constructs to Silver's specification language that allow extension writers to specify compile-time transformations such as dead-code elimination. It adds constructs to allow the extension writer to specify, within a host language or extension specification, how the control-flow graph is to be built, what conditions are to be checked (via formulas in the CTL logic), and which attributes are to be used in their evaluation. In the generated compilers' attribute grammar definitions, there are corresponding system calls to the external model-checker NuSMV. During the process of attribute evaluation that takes place when the source code is compiled, the model-checker is called with the control-flow graph model and the appropriate temporal logic formula as inputs. The results of running the checker are retrieved and can be used in further attribute evaluation, such as for the construction of optimized code.

The sections below describe the parts of the extension that variously provide mechanisms to specify how the control-flow graph is to be built from the program syntax tree, and to specify data-flow conditions as temporal logic formulas. The use of these constructs is illustrated using fragments of a Silver specification that adds compile-time dead code elimination to the *C–* language specification.

A.3.1 Specifying the Control-Flow Graph

This section describes the constructs added to the Silver specification language to specify how program control-flow graphs are to be constructed during compilation. Figure A.6 shows a part of the Silver specification that does this for programs in the *C–* language. Defining the structure of the control-flow graph requires specifying how the graph's nodes, edges and node labels are to be generated.

Non-terminals that are the symbols of syntax tree nodes with corresponding control-flow graph nodes are flagged using the `cfg nodes` construct. In this example, the non-terminals `Stmt` and `Expr` are flagged. Note that not all statements and expression nodes will have corresponding graph nodes. For example, declaration statements do not have corresponding graph nodes. Specific graph nodes are created using the syntax

`cfg` *syntax-node* `[` *successor-list* `]`

Each graph node is created from its corresponding syntax tree node and a list of successor graph nodes. Nodes have the non-terminal type `CFG_Node`, the declaration of which is

```
synthesized attribute def :: String occurs on { Stmt, Expr };
synthesized attribute uses :: [ String ] occurs on { Stmt, Expr };

aspect production skip
stmt::Stmt ::= {
     stmt.def = "";
     stmt.uses = [ ];
}

aspect production assignment
assign::Stmt ::= var::Id_t '=' expr::Expr ';' {
     assign.def = var.lexeme;
     assign.uses = expr.uses;
}

aspect production reference
expr::Expr ::= var::Id_t {
     expr.def = "";
     expr.uses = [ var.lexeme ];
}

aspect production int_constant
expr::Expr ::= int::IntConstant_t {
     expr.def = "";
     expr.uses = [ ];
}

aspect production equality
equal::Expr ::= lhs::Expr '==' rhs::Expr {
     equal.def = "";
     equal.uses = lhs.uses ++ rhs.uses;
}

aspect production plus
sum::Expr ::= expr1::Expr '+' expr2::Expr {
     sum.def = "";
     sum.uses = expr1.uses ++ expr2.uses;
}
```

Figure A.5: Specifying the labels on the nodes of a control-flow graph.

automatically added by the extension.

The control-flow graph edges are defined by each node's successor list. Here, these lists are computed in aspect productions which define the synthesized attribute `entry` and the inherited attribute `successor`. For a syntax node, these attributes specify the entry node and successor node, respectively, of the sub-graph corresponding to the syntax node's sub-tree. In Figure A.6, the `if_then_else` sub-tree has a corresponding control-flow sub-graph in which the entry node is a new node constructed from the conditional expression's syntax node. This entry node has the `entry` nodes of the `then` and `else` clauses as its immediate successors. While a control-flow graph node may have multiple successor nodes, the sub-graph corresponding to a syntax tree node's sub-tree has only one successor node. The successor of the `if_then_else` sub-graph as a whole is passed to its children. It will be used during the creation of the other nodes in its sub-graph.

Certain of the attributes are flagged as control-flow graph label attributes using the `cfg attributes` construct. Their attribute instance values are used to generate the node labels. Control-flow graph nodes are labeled with the evaluated values of these attributes on their corresponding syntax tree nodes. Here, both `def` and `uses` are flagged as label attributes. If a non-terminal is declared to be a control-flow graph node non-terminal, it must define the values of any label attributes that decorate it.

A.3.2 Model-Checking CTL Formulas with NuSMV

Our extension adds constructs for specifying data-flow properties as CTL formulas and for checking them at compile time by calling the model-checker NuSMV [59] on a model constructed from the program's control-flow graph. The results of the calls are available for use within attribute definitions to generate optimized code. As shown in Figure A.7, the NuSMV model for a program's control-flow graph is constructed using the syntax `smvmodel` *CFG-entry-node state-variable-range-list*.

NuSMV models have the non-terminal type `SMV_Model`, the declaration of which is automatically added by the extension. NuSMV, like other finite state checkers, requires the ranges of possible values for all state variables in any formula to be checked, in this case the label attributes `def` and `uses`. The `ranges over` and `ranges over powerset of` constructs are used to define the ranges of `def` and `uses` as the set of program variables, and that set's power-set, respectively. We have elided the definitions of the attribute `allVariables` which at the root node, contains all the variables in the program. The model is defined on the root and passed down via the inherited attribute `smvModel`.

The `|=` construct is used to evaluate the results of model-checking a NuSMV CTL formula against a particular node in a model, using the syntax

NuSMV-model `,` *CFG-node* `|=` *SMV-formula*

The assignment production uses the `|=` construct to model-check the CTL formula that states the condition under which the assignment is dead. The result of running the model-checker on the assignment node is used to perform a simple optional optimization that replaces the assignment with a `skip` statement if it is dead. The optimized version of

```
cfg nodes [ Stmt, Expr ];
synthesized attribute entry :: CFG_Node occurs on { Stmt, Stmts };
inherited attribute successor :: CFG_Node occurs on { Stmt, Stmts };
cfg attributes [ def, uses ];

aspect production sequence
stmts::Stmts ::= first::Stmt rest::Stmts {
     stmts.entry = first.entry;
     first.successor = rest.entry;
     rest.successor = stmts.successor;
}

aspect production stmt_one
stmts::Stmts ::= stmt::Stmt {
     stmts.entry = stmt.entry;
     stmt.successor = stmts.successor;
}

aspect production if_then_else
if::Stmt ::= 'if' '(' cond::Expr ')' then::Stmts 'else' else::Stmts {
     if.entry = cfg cond [ then.entry, else.entry ];
     then.successor = if.successor;
     else.successor = if.successor;
}

aspect production while
loop::Stmt ::= 'while' '(' cond::Expr ')' body::Stmts {
     loop.entry = cfg cond [ body.entry, loop.successor ];
     body.successor = loop.entry;
}

aspect production assignment
assign::Stmt ::= var::Id_t '=' expr::Expr ';' {
     assign.entry = cfg assign [ assign.successor ];
}

aspect production declaration
dcl::Stmt ::= type::Type var::Id_t ';' { dcl.entry = dcl.successor; }

aspect production skip
stmt::Stmt ::= { stmt.entry = cfg stmt [ stmt.successor ]; }
```

Figure A.6: Specifying the construction of control-flow graphs for $C-$ programs.

```
inherited attribute smvModel :: SMV_Model occurs on { Stmts, Stmt };

aspect production program_root
root::Root ::= stmts::Stmt {
  stmts.smvModel = smvmodel stmts.entry
                   [ def ranges over root.allVariables,
                     uses ranges over powerset of root.allVariables ];
}

synthesized attribute optimizedStmt :: Stmt occurs on Stmt;
synthesized attribute optimizedStmts :: Stmts occurs on Stmts;

aspect production assignment
assign::Stmt ::= var::Id expr::Expr {
  assign.optimizedStmt = if assign.smvModel, assign.entry |=
          AX (A [ !(var.lexeme in uses)
              U (def == var.lexeme && !(var.lexeme in uses)) ]
            || AG !(var.lexeme in uses))
     then skip () else assign;
}

aspect production sequence
stmts::Stmts ::= first::Stmt rest::Stmts {
  stmts.optimizedStmts = sequence(first.optimizedStmt,rest.optimizedStmts);
}
```

Figure A.7: Performing dead-code elimination on a *C*– program by model-checking the NuSMV representation of its control-flow graph.

the *C*– program is constructed using the `optimizedStmt` and `optimizedStmts` attributes.

A.4 Discussion

The data-flow analysis extension to Silver described in this appendix aids in the writing of high-level language specifications in which both syntax-directed and control-flow based analyses are performed during semantic analysis. As described in previous chapters, an important goal of extensible language frameworks such as Silver is facilitating the writing of extension specifications by domain-experts, who may not have expertise in the writing of compilers. It is desirable, therefore, for Silver to include a high-level formalism such as CTL that allows for writing data-flow conditions that can be used for domain-specific optimizations. Allowing the extension writer to specify, within the same declarative framework, complex semantic analyses on both the program syntax tree (using traditional attributes) and the program control-flow graph (using temporal logic formulas) makes Silver more useful. Further, the approach described here can be used to extend Silver with analyses based on other temporal logics and model-checkers as they are developed and found to be useful for specifying language extensions.

Appendix B

Attribute Grammar Examples

This appendix gives three examples of higher-order attribute grammars. It also gives examples of the process of local tree creation for these grammars during higher-order attribute evaluation. It also lists the rules constructed for each grammar by the termination analysis in Section 5.5 that attempt to model this process of local tree creation.

- Section B.1 describes G_1, a grammar that specifies a small imperative language and uses local attributes to provide a simple simulation of forwarding. The termination analysis in Chapter 5 can be used to show that evaluation terminates for this grammar.
- Section B.2 describes G_2, another grammar for a small imperative language but one that uses inherited attributes to implement the program environment. The termination analysis can be used to show that evaluation terminates for this grammar.
- Section B.3 describes G_3, an extended version of G_1. It defines additional attributes to construct a simple "host language translation" for each program in the grammar, in which sub-trees are replaced by the trees they forward to. Since the rules generated for this grammar are non-terminating, the termination analysis cannot be used to show that attribute evaluation always terminates for this grammar.

B.1 G_1: A Simple Grammar that Simulates Forwarding

Our first example grammar G_1 implements a simple imperative language. An example of a program written in this grammar is shown in Figure B.1. Figure B.2 gives the Silver specification for G_1. The grammar includes non-terminal and production declarations that define statements, expressions and types. Terminal declarations have been elided in this and other figures in this appendix, but are similar to those in Figure 3.2. It computes a single pretty-print attribute `pp` on its productions. There are no inherited

```
boolean x;
int y;
int z;
if (x)
   do y = z;
      while (x);
if (x)
   do z = y;
      while (x);
```

Figure B.1: A sample program in the grammar G_1.

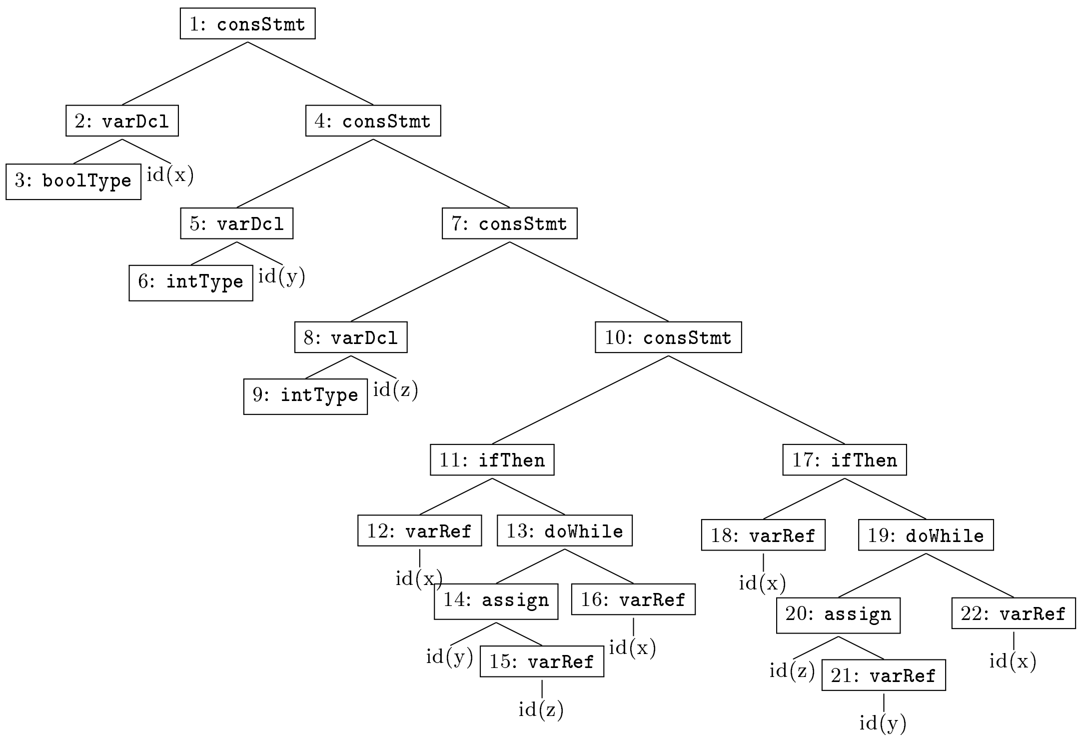
attributes. G_1 uses local attributes to simulate forwarding. For example, the `ifThen` production defines its `pp` attribute in terms of a local tree that is constructed as an equivalent `ifThenElse` tree. Similarly, the `doWhile` production is defined in terms of the `while` production.

B.1.1 An Example of Higher-Order Attribute Evaluation

This section gives an example of the process of local tree creation during the attribution of a tree in G_1. We assume that attribute evaluation takes place for the syntax tree T_0 below, which corresponds to the program in Figure B.1.

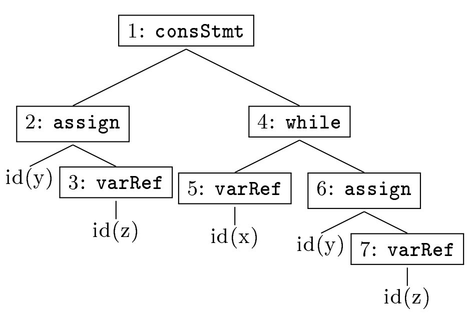
```
start nonterminal Stmt;
nonterminal Expr, Type;

synthesized attribute pp :: String occurs on Stmt, Expr, Type;

concrete production varDcl s::Stmt ::= t::Type id::Id_t ';' {
  s.pp = t.pp ++ " " ++ id.lexeme ++ ";";
}

concrete production assign s::Stmt ::= id::Id_t '=' e::Expr ';' {
  s.pp = id.lexeme ++ " = " ++ e.pp ++ ";";
}

concrete production while s::Stmt ::= 'while' '(' e::Expr ')' s1::Stmt {
  s.pp = "while ( " ++ e.pp ++ " ) " ++ s1.pp;
}
concrete production ifThenElse
s::Stmt ::= 'if' '(' e::Expr ')' s1::Stmt 'else' s2::Stmt {
  s.pp = "if ( " ++ e.pp ++ " ) " ++ s1.pp ++ " else " ++ s2.pp;
}

concrete production emptyStmt s::Stmt ::= ';' { s.pp = ";"; }

concrete production consStmt s::Stmt ::= s1::Stmt s2::Stmt {
  s.pp = s1.pp ++ s2.pp;
}
- The do-while statement is defined in terms of the while statement.
concrete production doWhile
s::Stmt ::= 'do' s1::Stmt 'while' '(' e::Expr ')' ';' {
  s.pp = "do " ++ s1.pp ++ " while ( " ++ e.pp ++ " );";
  local attribute fs::Stmt = consStmt (s1, while ('while', e, s1));
}
- The if-then statement is defined in terms of the if-then-else statement.
concrete production ifThen s::Stmt ::= 'if' '(' e::Expr ')' s1::Stmt {
  s.pp = "if ( " ++ e.pp ++ " ) " ++ s1.pp;
  local attribute fs::Stmt = ifThenElse ('if', e, s1, 'else', emptyStmt());
}

concrete production varRef e::Expr ::= id::Id_t { e.pp = id.lexeme; }
concrete production boolType t::Type ::= 'boolean' { t.pp = "boolean"; }
concrete production intType t::Type ::= 'int' { t.pp = "int"; }
```

Figure B.2: G_1: A grammar that defines a simple imperative language and simulates forwarding with local attributes.

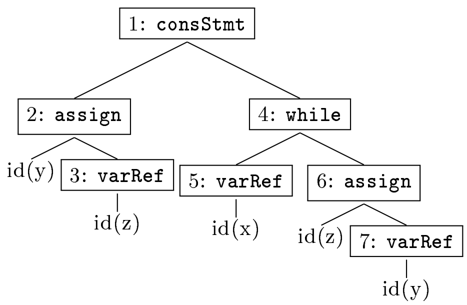


In these tree representations non-terminal nodes are labeled with numerical identifiers. We assume that attribute instances are evaluated in the sequence shown below. We have omitted steps that evaluate primitive attributes and non-local higher-order steps that evaluate tree terms without creating new syntax trees.

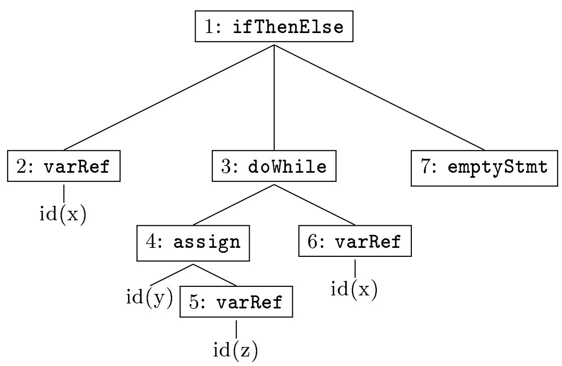
The first local tree creating step occurs on node 13 in the program syntax tree T_0, which corresponds to the first `doWhile` statement in the program. It constructs its equivalent `while` tree T_1 as the value of the local attribute `fs`.



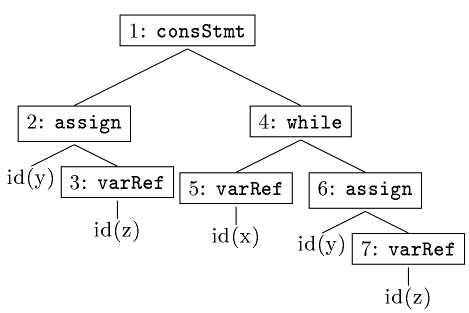
The second local tree creating step occurs on node 19 in the program syntax tree T_0, which corresponds to the second `doWhile` statement in the program. It constructs its equivalent `while` tree T_2 as the value of the local attribute `fs`.



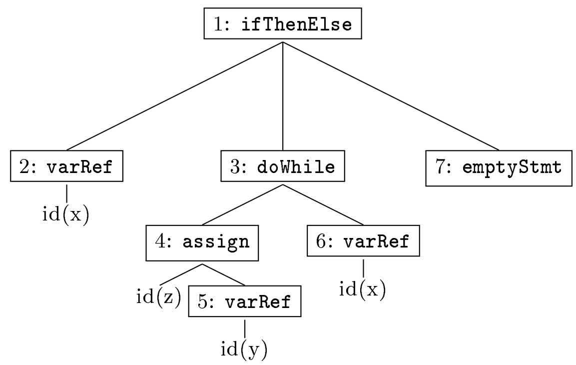
The third local tree creating step occurs on node 11 in the program syntax tree T_0, which corresponds to the first `ifThen` statement in the program. It constructs its equivalent `ifThenElse` tree T_3 as the value of the local attribute `fs`.



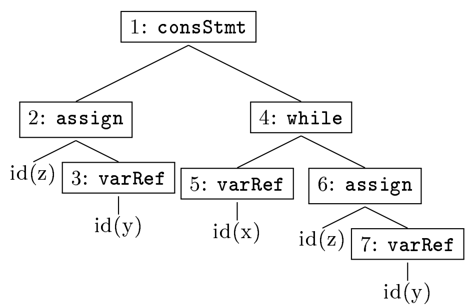
The fourth local tree creating step occurs on node 3 in the tree T_3 created in the previous step, which corresponds to a `doWhile` statement. It constructs its equivalent `while` tree T_4 as the value of the local attribute `fs`.



The fifth local tree creating step occurs on node 17 in the program syntax tree T_0, which corresponds to the second `ifThen` statement in the program. It constructs its equivalent `ifThenElse` tree T_5 as the value of the local attribute `fs`.



The sixth and final local tree creating step occurs on node 3 in the tree T_5 created in the previous step, which corresponds to a `doWhile` statement. It constructs its equivalent `while` tree T_6 as the value of the local attribute `fs`.



For this evaluation sequence, the tree of locals (TOL) would be constructed as shown in Figure B.3. The figure shows the structure of the tree of locals after each tree creation step listed above. The root of the tree of locals is the program syntax tree T_0. Each node of a tree of local's representation specifies the syntax tree and the node of its parent tree on which it was created. A path in the tree of locals defines a tree creation sequence. The trees in the tree creation sequence T_0, T_3, T_4 are the original program syntax tree, followed by an `ifThenElse` tree, and finally, the tree forwarded to by the `doWhile` tree.

B.1.2 Constructing Rewrite Rules to Model Tree Creation

The rules generated for the grammar G_1 are terminating and are as follows:

- `doWhile` $(s_1,\ e) \longrightarrow$ `consStmt` $(s_1,$ `while` $(e,\ s_1))$
- `ifThen` $(e,\ s_1) \longrightarrow$ `ifThenElse` $(e,\ s_1,$ `emptyStmt` $())$

Consider the tree creation sequence T_0, T_3, T_4 in the tree of locals in Figure B.3. These rules can be used to derive each of these trees from the sub-tree of its parent on which it was created, as follows:

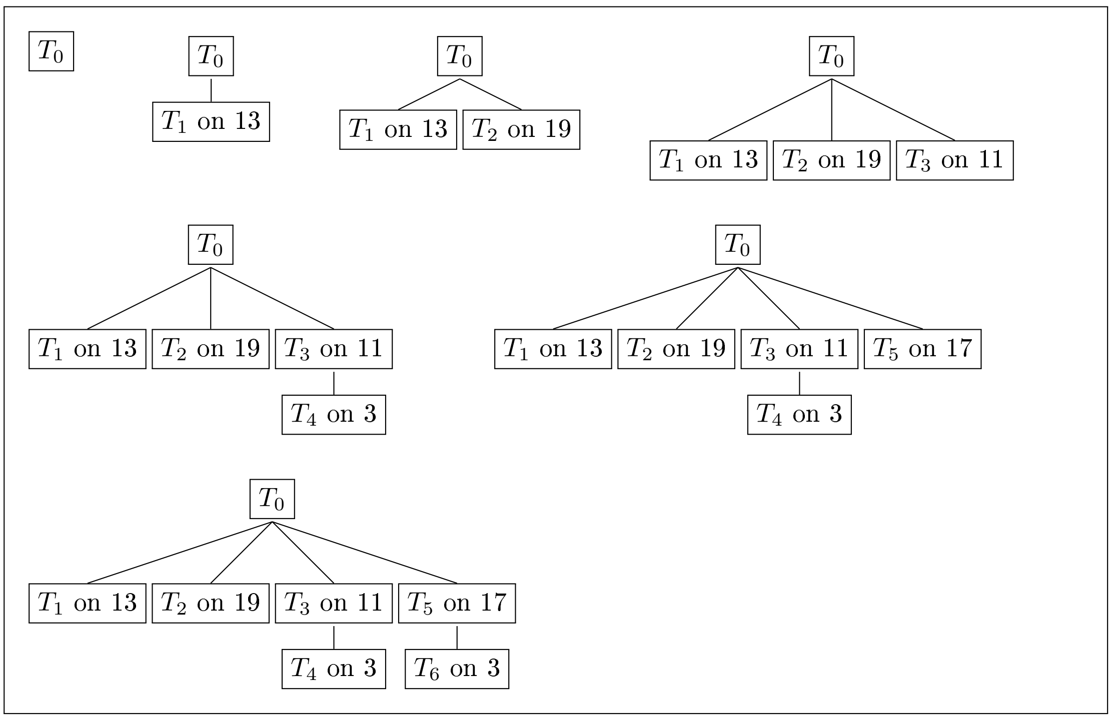
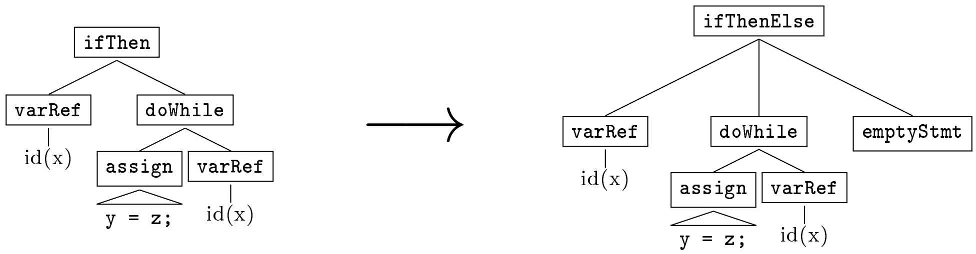
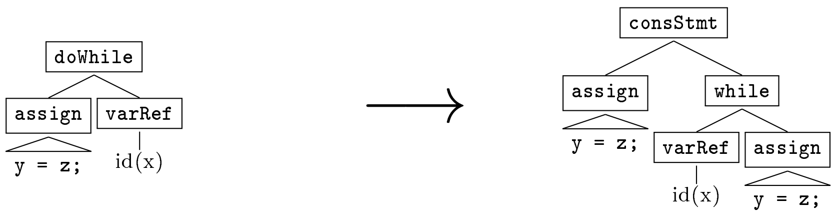
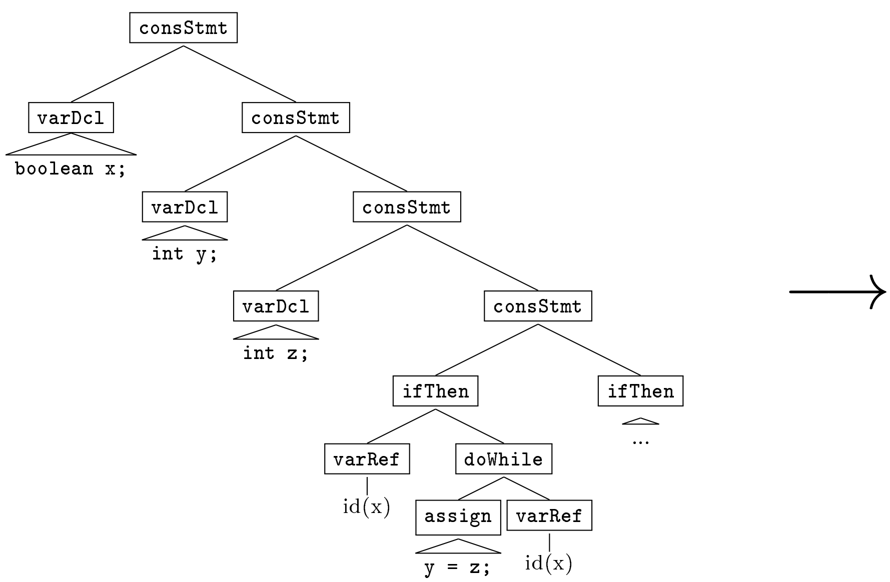


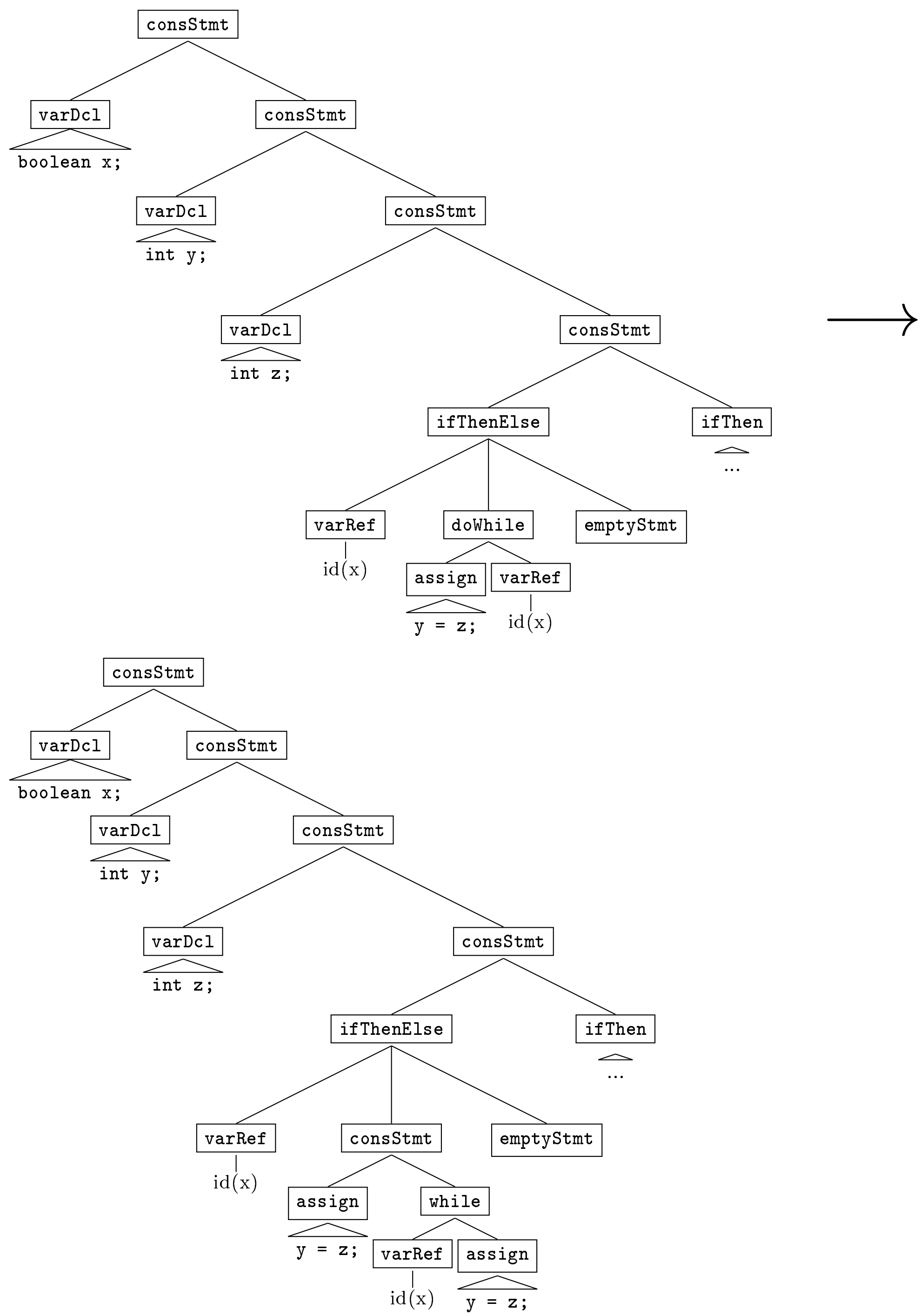
Figure B.3: The steps in the construction of the tree of locals corresponding to a higher-order evaluation sequence for the grammar G_1.





Since each local tree's parent tree is a sub-term of its predecessor in the tree creation sequence, we can generate a rewrite sequence corresponding to the entire tree creation sequence as follows:




consStmt
varDcl
boolean x;
consStmt
varDcl
int y;
consStmt
varDcl
int z;
consStmt
ifThenElse
ifThen
...
varRef
id(x)
doWhile
emptyStmt
assign
y = z;
varRef
id(x)
consStmt
varDcl
boolean x;
consStmt
varDcl
int y;
consStmt
varDcl
int z;
consStmt
ifThenElse
ifThen
...
varRef
id(x)
consStmt
emptyStmt
assign
y = z;
while
varRef
id(x)
assign
y = z;

B.2 G_2: A Grammar with Multiple Levels of Inherited Attributes

```
typedef int t1;
typedef boolean t2;
int x1;
t2 x2;
x1 = x2;
```

Figure B.4: A sample program in the grammar G_2.

Our second example grammar G_2 specifies a simple imperative language, using inherited attributes to implement the program environment. An example of a program in this grammar is shown in Figure B.4. Figures B.5, B.6 and B.7 give the Silver specification for G_2. The specification defines productions to construct lists of bindings of variables to types. It further defines a second level of variable bindings by defining a type environment which maps identifiers to types. These type identifiers can then be used in variable declarations. The grammar defines several synthesized and inherited attributes that construct the variable and type environments and pass them around the program syntax tree for use in computing error messages on each node. This grammar provides an example of inherited attributes decorating non-terminals that are themselves the types of inherited attributes. To this end, it contains productions such as `singleEnv` and `singleTypeEnv` which are redundant, and excludes a simple `Type` ::= $\mathtt{Id}_{\mathtt{t}}$ production.

B.2.1 An Example of Higher-Order Attribute Evaluation

This section gives an example of the process of local tree creation during the attribution of a tree in G_2. We assume that attribute evaluation takes place for the syntax tree T_0 below which corresponds to the program in Figure B.4.

```
start nonterminal Root;
nonterminal Stmt, Expr, Type, TypeRep, Env, TypeEnv;

inherited   attribute env      :: Env        occurs on Stmt, Expr;
synthesized attribute defs     :: Env        occurs on Stmt;
- TypeEnv is an inherited attribute that decorates a non-terminal Env
- that is itself the type of an inherited attribute.
inherited   attribute typeEnv  :: TypeEnv    occurs on Stmt, Expr, Env;
synthesized attribute typeDefs :: TypeEnv    occurs on Stmt;
synthesized attribute errors   :: [ String ] occurs on Root, Stmt, Expr;
inherited   attribute lookFor  :: String  occurs on Env, TypeEnv;
synthesized attribute found    :: Boolean occurs on Env, TypeEnv;
synthesized attribute typeRep  :: TypeRep occurs on Type, Env, TypeEnv;
synthesized attribute name     :: String  occurs on TypeRep;

concrete production root r::Root ::= s::Stmt {
  r.errors  = s.errors;
  s.env     = emptyEnv ();  s.typeEnv = emptyTypeEnv ();
}

concrete production emptyStmt s::Stmt ::= ';' {
  s.defs     = emptyEnv ();  s.typeDefs = emptyTypeEnv ();
}

concrete production consStmt s::Stmt ::= s1::Stmt s2::Stmt {
  s.defs     = appendEnv (s1.defs, s2.defs);
  s.typeDefs = appendTypeEnv (s1.typeDefs, s2.typeDefs);
  s.errors   = s1.errors ++ s2.errors;
  s1.env     = s.env;  s1.typeEnv = s.typeEnv;
  s2.env     = appendEnv (s1.defs, s.env);
  s2.typeEnv = appendtypeEnv (s1.typeDefs, s.typeEnv);
}
- The typedef statement associates types with type identifiers.
concrete production typeDcl s::Stmt ::= 'typedef' te::Type id::Id_t ';' {
  s.defs     = emptyEnv ();
  s.typeDefs = consTypeEnv (id, te.typeRep, emptyTypeEnv ());
}
concrete production varDclType s::Stmt ::= te::Type id::Id_t ';' {
  s.defs     = consEnvType (id, te.typeRep, emptyEnv ());
  s.typeDefs = emptyEnv ();
}
```

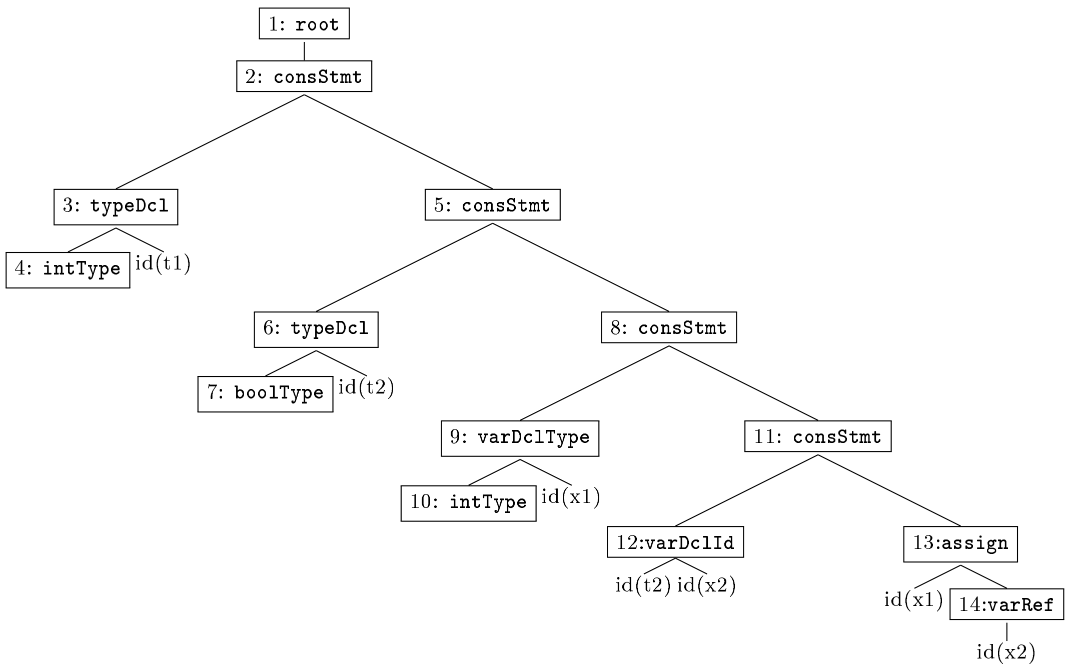
Figure B.5: G_2: A grammar with inherited attributes decorating non-terminals that are themselves the types of inherited attributes, part 1 of 3.

```
- Type identifiers can be used to declare regular variables.
concrete production varDclId s::Stmt ::= tid::Id_t id::Id_t ';' {
  s.defs     = consEnvId (id, tid, emptyEnv ());
  s.typeDefs = emptyEnv ();
}

concrete production assign s::Stmt ::= id::Id_t '=' e::Expr ';' {
  s.defs     = emptyEnv ();  s.typeDefs = emptyTypeEnv ();
  e.env      = s.env;  e.typeEnv  = s.typeEnv;
}
- Variable references perform environment look-ups.
concrete production varRef e::Expr ::= id::Id_t {
  local attribute ee::Env = e.env;
  ee.lookFor = id.lexeme;  ee.typeEnv = e.typeEnv;
  e.errors   = if ee.found then [ ]
               else [ "Undeclared identifier: " ++ id.lexeme ];
}
concrete production boolType te::Type ::= 'boolean' {
  te.typeRep = boolTypeRep ();
}
concrete production intType te::Type ::= 'int' {te.typeRep = intTypeRep();}
abstract production boolTypeRep tr::TypeRep ::= { tr.name = "boolean"; }
abstract production intTypeRep tr::TypeRep ::= { tr.name = "int"; }
- Adding a variable binding to a pre-existing environment.
- The production forwards to an equivalent tree constructed using
- the singleEnv and appendEnv productions.
abstract production consEnvType e::Env ::= id::Id_t t::TypeRep rest::Env {
  local attribute fe::Env = appendEnv (singleEnv (id, t), rest);
  fe.lookFor = e.lookFor;
  e.found    = fe.found;  e.typeRep  = fe.typeRep;
}
- Adding a variable binding (declared using a type identifier
- to a pre-existing environment. The type identifier is resolved and
- the production forwards to an equivalent consEnvType tree.
abstract production consEnvId e::Env ::= id::Id_t tid::Id_t rest::Env {
  local attribute te::TypeEnv = e.typeEnv;
  te.lookFor = tid.lexeme;
  local attribute fe::Env = consEnvType (id, te.typeRep, rest);
  fe.lookFor = e.lookFor;  fe.typeEnv = e.typeEnv;
  e.found    = fe.found;  e.typeRep  = fe.typeRep;
}
```

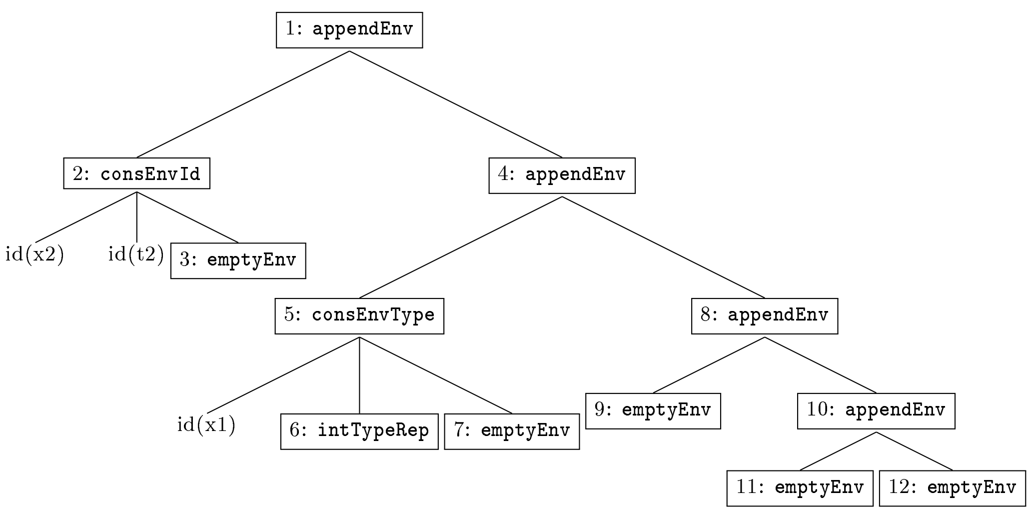
Figure B.6: G_2: A grammar with inherited attributes decorating non-terminals that are themselves the types of inherited attributes, part 2 of 3.

```
abstract production singleEnv e::Env ::= id::Id_t t::TypeRep {
  e.found    = id.lexeme == e.lookFor;
  e.typeRep  = if id.lexeme == e.lookFor then t
               else error ("Unknown id " ++ e.lookFor);
}

abstract production emptyEnv e::Env ::= {
  e.found   = false;  e.typeRep = error ("Unknown id " ++ e.lookFor);
}

abstract production appendEnv e::Env ::= e1::Env e2::Env {
  e.found    = e1.found || e2.found;
  e.typeRep  = if e1.found then e1.typeRep else e2.typeRep;

  e1.lookFor = e.lookFor;  e1.typeEnv = e.typeEnv;
  e2.lookFor = e.lookFor;  e2.typeEnv = e.typeEnv;
}
- Adding a type variable binding to a pre-existing type environment.
- The production forwards to an equivalent tree constructed using
- the singleTypeEnv and appendTypeEnv productions.
abstract production consTypeEnv
te::TypeEnv ::= tid::Id_t t::TypeRep rest::TypeEnv {
  local attribute fte::TypeEnv = appendTypeEnv(singleTypeEnv(tid, t),rest);
  fte.lookFor = te.lookFor;
  te.found    = fte.found; te.typeRep  = fte.typeRep;
}

abstract production singleTypeEnv te::TypeEnv ::= tid::Id_t t::TypeRep {
  te.found    = tid.lexeme == te.lookFor;
  te.typeRep  = if tid.lexeme == te.lookFor then t
                else error ("Unknown id " ++ te.lookFor);
}

abstract production emptyTypeEnv te::TypeEnv ::= {
  te.found    = false;  te.typeRep  = error ("Unknown id " ++ te.lookFor);
}
abstract production appendTypeEnv te::TypeEnv::= te1::TypeEnv te2::TypeEnv{
  te.found    = te1.found || te2.found;
  te.typeRep  = if te1.found then te1.typeRep else te2.typeRep;
  te1.lookFor = te.lookFor; te2.lookFor = te.lookFor;
}
```

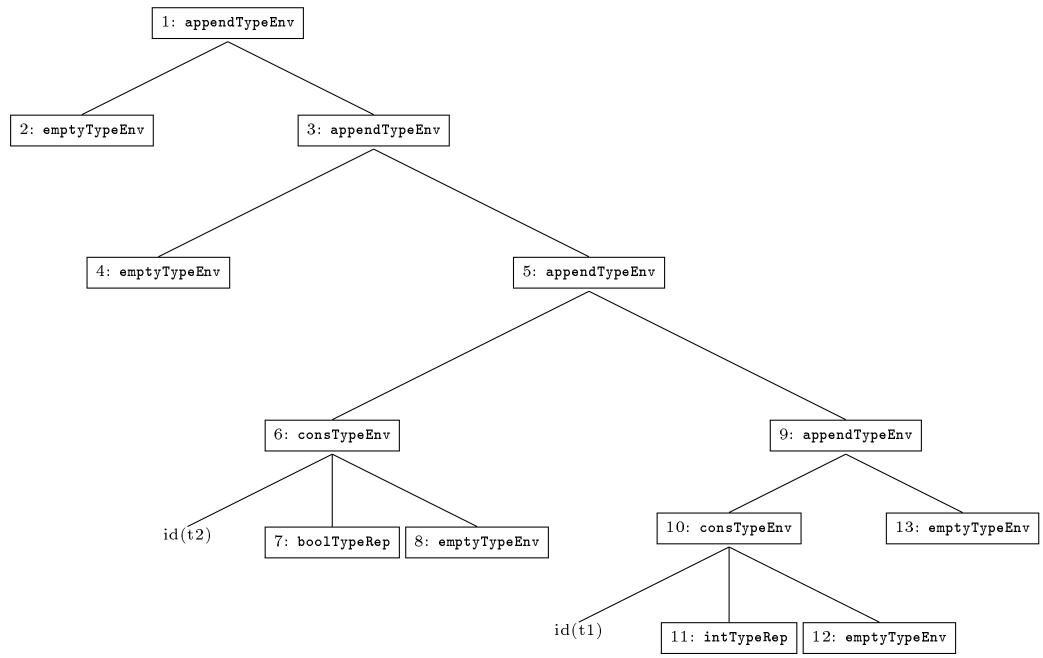
Figure B.7: G_2: A grammar with inherited attributes decorating non-terminals that are themselves the types of inherited attributes, part 3 of 3.



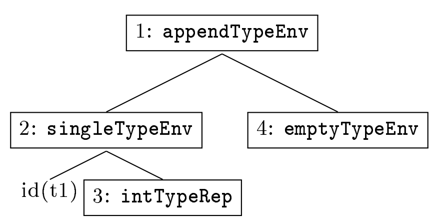
The first local tree creating step occurs on node 14 in the program syntax tree T_0, which corresponds to the variable reference in the program's assignment statement. It constructs its environment tree T_1 as the value of the local attribute `ee`.



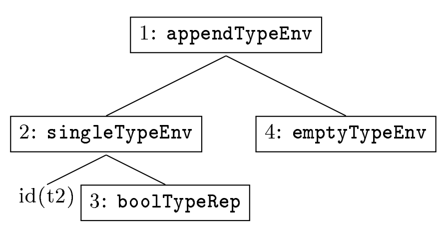
The second local tree creating step occurs on node 2 in the environment tree T_1 created in the previous step. This node's production `consEnvId` constructs its type environment tree T_2 as the value of the local attribute `te`.



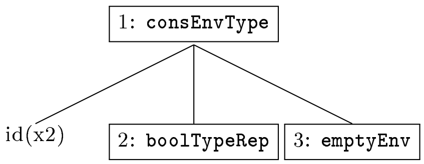
The third local tree creating step occurs on node 10 in the type environment tree T_2 created in the previous step. This node's production `consTypeEnv` constructs its equivalent `appendTypeEnv` tree T_3 as the value of the local attribute `fte`.



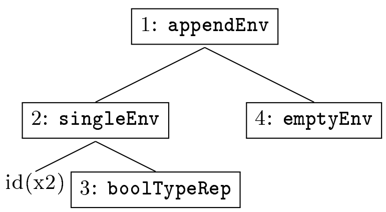
The fourth local tree creating step occurs on node 6 in the type environment tree T_2. This node's production `consTypeEnv` constructs its equivalent `appendTypeEnv` tree T_4 as the value of the local attribute `fte`.



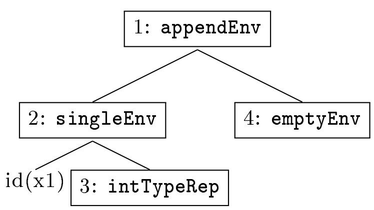
The fifth local tree creating step occurs on node 2 in the environment tree T_1. This node's production `consEnvId` constructs its equivalent `consEnvType` tree T_5 as the value of the local attribute `fe`.



The sixth local tree creating step occurs on node 1 of the tree T_5 created in the previous step. The node's production `consEnvType` constructs its equivalent `appendEnv` tree T_6 as the value of the local attribute `fe`.



The seventh and final tree creating step occurs on node 5 in the environment tree T_1. This node's production `consEnvType` constructs its equivalent `appendEnv` tree T_7 as the value of the local attribute `fe`.



For this evaluation sequence, the tree of locals (TOL) would be constructed as shown in Figure B.8. The figure shows the structure of the tree of locals after each tree creation step listed above. The root of the tree of locals is the program syntax tree T_0. Each node of a tree of local's representation specifies the syntax tree and the node of its parent tree on which it was created.

B.2.2 Constructing Rewrite Rules to Model Tree Creation

The rules generated for the grammar G_2 are terminating and are as follows:

- `root` $(s) \longrightarrow s$
- `root` $(s) \longrightarrow$ `emptyEnv`
- `root` $(s) \longrightarrow$ `emptyTypeEnv`

- `emptyStmt` $\longrightarrow$ `emptyEnv`

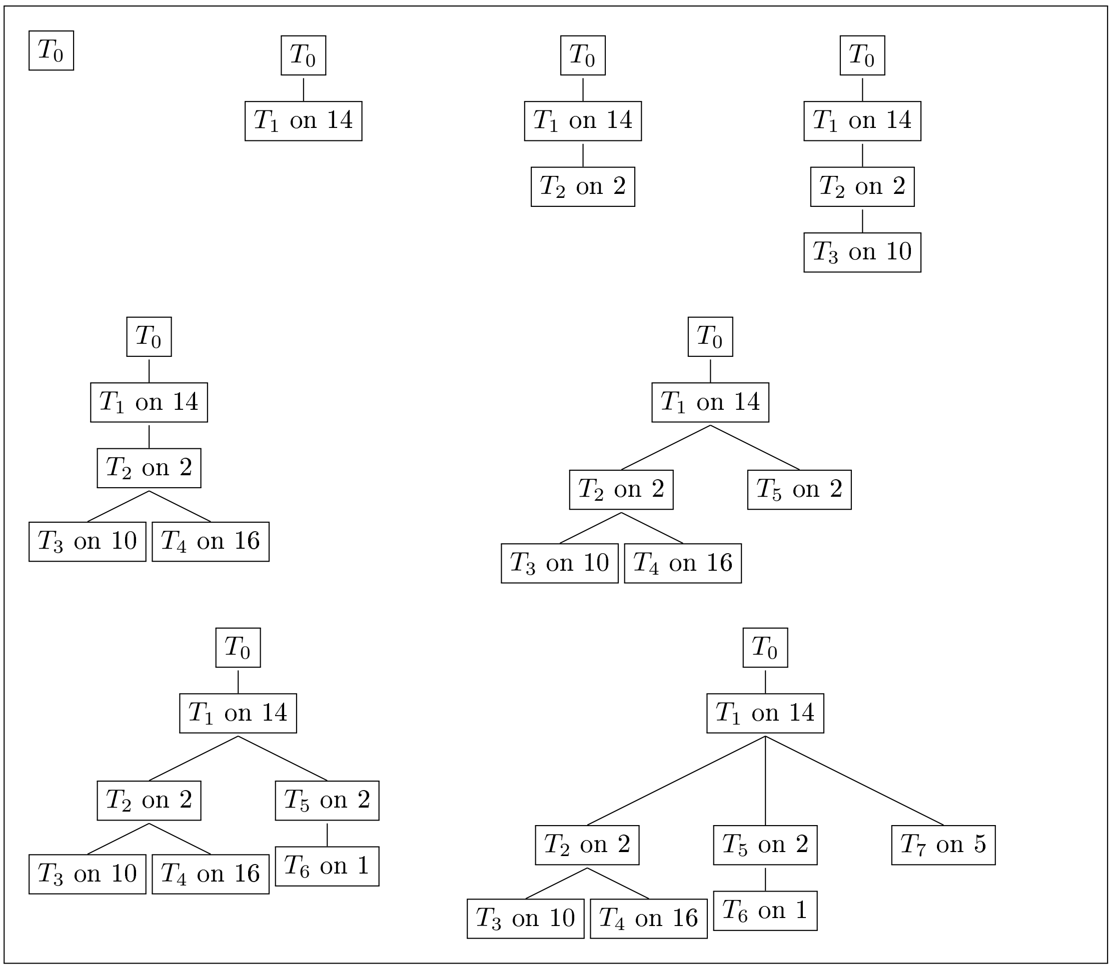


Figure B.8: The steps in the construction of the tree of locals corresponding to a higher-order evaluation sequence for the grammar G_2.

- emptyStmt $\longrightarrow$ emptyTypeEnv
- consStmt $(s_1,\ s_2) \longrightarrow$ appendEnv $(s_1,\ s_2)$
- consStmt $(s_1,\ s_2) \longrightarrow$ appendTypeEnv $(s_1,\ s_2)$
- consStmt $(s_1,\ s_2) \longrightarrow$ appendList $(s_1,\ s_2)$
- consStmt $(s_1,\ s_2) \longrightarrow$ INH
- consStmt $(s_1,\ s_2) \longrightarrow$ appendEnv $(s_1,$ INH$)$
- consStmt $(s_1,\ s_2) \longrightarrow$ appendTypeEnv $(s_1,$ INH$)$
- typeDcl $(te,\ id) \longrightarrow$ emptyEnv
- typeDcl $(te,\ id) \longrightarrow$ consTypeEnv $(id,\ te,$ emptyTypeEnv$)$
- varDclType $(te,\ id) \longrightarrow$ consEnvType $(id,\ te,$ emptyEnv$)$
- varDclType $(te,\ id) \longrightarrow$ emptyEnv
- varDclId $(tid,\ id) \longrightarrow$ consEnvId $(id,\ tid,$ emptyEnv$)$
- varDclId $(tid,\ id) \longrightarrow$ emptyEnv
- assign $(id,\ e) \longrightarrow$ emptyEnv
- assign $(id,\ e) \longrightarrow$ emptyTypeEnv
- assign $(id,\ e) \longrightarrow$ INH

- varRef $(id) \longrightarrow$ INH
- varRef $(id) \longrightarrow$ emptyList
- varRef $(id) \longrightarrow$ mkList (appendString (stringConstant, id))

- boolType $\longrightarrow$ boolTypeRep
- intType $\longrightarrow$ intTypeRep

- consEnvType $(id,\ t,\ rest) \longrightarrow$ appendEnv (singleEnv $(id,\ t)$, $rest$)
- consEnvId $(id,\ tid,\ rest) \longrightarrow$ INH
- consEnvId $(id,\ tid,\ rest) \longrightarrow$ consEnvType $(id,$ INH$,\ rest)$
- singleEnv $(id,\ t) \longrightarrow t$
- singleEnv $(id,\ t) \longrightarrow$ error (appendString (stringConstant, INH))
- emptyEnv $\longrightarrow$ error (appendString (stringConstant, INH))
- appendEnv $(e_1,\ e_2) \longrightarrow e_1$
- appendEnv $(e_1,\ e_2) \longrightarrow e_2$
- appendEnv $(e_1,\ e_2) \longrightarrow$ INH

- consTypeEnv $(tid,\ t,\ rest) \longrightarrow$ appendTypeEnv (singleTypeEnv $(tid,\ t)$, $rest$)
- singleTypeEnv $(tid,\ t) \longrightarrow t$
- singleTypeEnv $(tid,\ t) \longrightarrow$ error (appendString (stringConstant, INH))
- emptyTypeEnv $\longrightarrow$ error (appendString (stringConstant, INH))
- appendTypeEnv $(e_1,\ e_2) \longrightarrow e_1$

- `appendTypeEnv` $(e_1,\ e_2) \longrightarrow e_2$
- `appendTypeEnv` $(e_1,\ e_2) \longrightarrow$ `INH`

B.3 G_3: A Grammar with Non-Terminating Tree Creation

Our third example grammar G_3 is an extended version of G_1 and defines the same language. An example of a program in this language is shown in Figure B.1. Figures B.9 and B.10 show the Silver specification for G_3. In this specification, a new attribute `hostStmt` computes the "base" version of the program syntax, i.e., an equivalent version in which "extended" constructs such as `doWhile` and `ifThen` have been translated away to equivalent base constructs such as `while` and `ifThenElse`. In practical usage, such an attribute would only be computed once, off the initial program tree. In the evaluation model described in Section 2.3 however, the `hostStmt` attribute can be computed afresh on the base tree, and again on this new tree, and so on. Therefore tree creation is not terminating. The rules generated for this grammar (listed in the next section) cannot therefore be shown to be terminating.

B.3.1 Constructing Rewrite Rules to Model Tree Creation

The rules generated for the grammar G_3 are non-terminating and are as follows:

- `varDcl` $(t,\ id) \longrightarrow$ `varDcl` $(t,\ id)$
- `assign` $(id,\ e) \longrightarrow$ `assign` $(id,\ e)$
- `while` $(e,\ s_1) \longrightarrow$ `while` $(e,\ s_1)$
- `ifThenElse` $(e,\ s_1,\ s_2) \longrightarrow$ `ifThenElse` $(e,\ s_1,\ s_2)$
- `emptyStmt` $() \longrightarrow$ `emptyStmt` $()$
- `consStmt` $(s_1,\ s_2) \longrightarrow$ `consStmt` $(s_1,\ s_2)$
- `doWhile` $(s_1,\ e) \longrightarrow$ `consStmt` $(s_1,$ `while` $(e,\ s_1))$
- `ifThen` $(e,\ s_1) \longrightarrow$ `ifThenElse` $(e,\ s_1,$ `emptyStmt` $())$

```
start nonterminal Stmt;
nonterminal Expr, Type;

synthesized attribute pp :: String occurs on Stmt, Expr, Type;

- The base version of the program tree.
synthesized attribute hostStmt :: Stmt occurs on Stmt;

concrete production varDcl s::Stmt ::= t::Type id::Id_t ';' {
  s.pp = t.pp ++ " " ++ id.lexeme ++ ";";
  s.hostStmt = varDcl (t, id, ';');
}

concrete production assign s::Stmt ::= id::Id_t '=' e::Expr ';' {
  s.pp = id.lexeme ++ " = " ++ e.pp ++ ";";
  s.hostStmt = assign (id, '=', e, ';');
}

- In base constructs, hostStmt is constructed with the same production.
- The translation process is propagated down to child nodes.
concrete production while s::Stmt ::= 'while' '(' e::Expr ')' s1::Stmt {
  s.pp = "while ( " ++ e.pp ++ " ) " ++ s1.pp;
  s.hostStmt = while ('while', '(', e, ')', s1.hostStmt);
}

concrete production ifThenElse
s::Stmt ::= 'if' '(' e::Expr ')' s1::Stmt 'else' s2::Stmt {
  s.pp = "if ( " ++ e.pp ++ " ) " ++ s1.pp ++ " else " ++ s2.pp;
  s.hostStmt = ifThenElse ('if', '(', e, ')', s1.hostStmt,
                           'else', s2.hostStmt);
}

concrete production emptyStmt s::Stmt ::= ';' {
  s.pp = ";";
  s.hostStmt = emptyStmt (';');
}

concrete production consStmt s::Stmt ::= s1::Stmt s2::Stmt {
  s.pp = s1.pp ++ s2.pp;
  s.hostStmt = consStmt (s1.hostStmt, s2.hostStmt);
}
```

Figure B.9: G_3: A grammar for which tree creation does not terminate, part 1 of 2.

```
- For extended constructs, hostStmt is computed off the forwarded to tree.
concrete production doWhile
s::Stmt ::= 'do' s1::Stmt 'while' '(' e::Expr ')' ';' {
  s.pp = "do " ++ s1.pp ++ " while ( " ++ e.pp ++ " );";
  local attribute fs::Stmt = consStmt (s1, while ('while', e, s1));
  s.hostStmt = fs.hostStmt;
}

concrete production ifThen s::Stmt ::= 'if' '(' e::Expr ')' s1::Stmt {
  s.pp = "if ( " ++ e.pp ++ " ) " ++ s1.pp;
  local attribute fs::Stmt = ifThenElse ('if', e, s1, 'else', emptyStmt());
  s.hostStmt = fs.hostStmt;
}

concrete production varRef e::Expr ::= id::Id_t { e.pp = id.lexeme; }

concrete production boolType t::Type ::= 'boolean' { t.pp = "boolean"; }

concrete production intType t::Type ::= 'int' { t.pp = "int"; }
```

Figure B.10: G_3: A grammar for which tree creation does not terminate, part 2 of 2.

Appendix C

Proofs Related to the Termination Analysis

This appendix contains material related to the termination analysis described in Chapter 5. In Section C.1, we consider the subset of expressions that define the trees in tree creation steps, and describe the structure of inductive proofs on them. In Section C.2, we prove **Lemma** 6, which states that the generated rules can be used to model the parts of each tree creation sequence that do not access inherited attributes. We also state and prove a couple of auxiliary lemmas.

C.1 Inductive Proofs on Higher-Order Tree Creating Expressions

The proofs in this appendix are inductive proofs on the structure of evaluable higher-order expressions. In this section, we define the set of evaluable higher-order expressions and the structure of inductive proofs on him.

As described in Section 2.2, *Expr* is the type of expressions on the right-hand sides of attribute definitions. An expression e on a production p is a higher-order expression if $type_e(p,\ e) \in NT\ \cup\ T$. The set of higher-order expressions on a production p is given by the following grammar:

$$
\begin{array}{lll}
\mathit{Expr} ::= & & \\
 & \#i & \\
 | & \#0.a_I & (a_I \in NT\ \cup\ T) \\
 | & \#i.a_S & (a_S \in NT\ \cup\ T) \\
 | & l.a_S & (a_S \in NT\ \cup\ T) \\
 | & q(e_1,\ ...,\ e_{n_q}) & (type_e(p,\ e_i) \in NT\ \cup\ T \text{ for } 1 \leq i \leq n_q) \\
 | & c & \\
 | & \texttt{if } e_C \texttt{ then } e_T \texttt{ else } e_E & (type_e(p,\ e_T) = type_e(p,\ e_E) \in NT\ \cup\ T)
\end{array}
$$

Higher-order expressions include child tree references, terminal symbols and production calls (which to be type-correct must have arguments that are also higher-order expressions). Also included are conditional expressions, in which both sub-expressions are higher-order expressions. Function calls are not present, except in the conditions of conditional expressions. Attribute accesses of non-terminal or terminal types are included.

$dep : N \longrightarrow Expr \longrightarrow Attribution \longrightarrow \mathcal{P}(Instance)$ returns the set of attribute instances required to evaluate a given higher-order, evaluable expression on a given node during evaluation.

For an expression e on a node n where $type_e(prod(n),\ e) \in NT\ \cup\ T$ and $\forall (n'\#a') \in dep(n,\ e,\ \Gamma)\ .\ \Gamma[n'\#a'] \neq \bot,\ dep(n,\ e,\ \Gamma)$ is defined as follows:

$$
\begin{array}{ll}
dep(n,\ \#i,\ \Gamma) = \{\ \} & \text{(parent or child tree)} \\
dep(n,\ \#0.a_I,\ \Gamma) = \{n\#a_I\} & \text{(inh. occurrence on parent)} \\
dep(n,\ \#i.a_S,\ \Gamma) = \{child(n,i)\#a_S\} & \text{(syn. occurrence on child)} \\
dep(n,\ l.a_S,\ \Gamma) = \{(\Gamma[n\#l])\#a_S\} & \text{(syn. occurrence on local)} \\
 & \\
dep(n,\ q(e_1, ...,\ e_{n_q}),\ \Gamma) = \bigcup_{i=1}^{n_q} dep(n,\ e_i,\ \Gamma) & \text{(tree creation)} \\
dep(n,\ c,\ \Gamma) = \{\ \} & \text{(terminal symbol)} \\
dep(n,\ \texttt{if}\ e_C\ \texttt{then}\ e_T\ \texttt{else}\ e_E,\ \Gamma) & \\
\quad = \begin{cases} dep(n,\ e_T,\ \Gamma) & \text{if } eval(n,\ e_C,\ \Gamma) = \texttt{true} \\ dep(n,\ e_E,\ \Gamma) & \text{otherwise} \end{cases} & \text{(conditional expression)}
\end{array}
$$

Figure C.1: The set of attribute instances required to evaluate a given higher-order, evaluable expression.

An evaluable higher-order expression on a production is any valid expression on the right-hand side of any of its definitions such that all of its required attribute instances are defined. Some of the proofs in this appendix assume that a particular property is true for all the required attribute occurrences of a given higher-order evaluable expression. Any evaluable higher-order expression is such that all its required attribute instances are also higher-order. This is obvious for all higher-order expressions except conditional expressions. As shown in Figure C.1, the set of required attribute instances for a higher-order conditional expression may include non-higher order attribute instances, viz., those present in its condition. However, if the conditional expression is evaluable, then the set of required attribute instances is defined to be only the required attribute instances of the appropriate sub-expression, which are all higher-order. The required attribute instances for a synthesized attribute access off a local attribute are similarly defined to include only the synthesized attribute instance, if the expression is evaluable.

We now briefly describe the structure of inductive proofs on evaluable higher-order expressions. To show inductively that a property P holds on all evaluable higher-order expressions on a production, we show that P holds on all base cases and on all inductive cases. The base cases are as follows:

- The expression is a signature variable.
- The expression is a terminal symbol.
- The expression is a synthesized attribute access on a child.
- The expression is an inherited attribute access.

In each inductive case, we show that P holds on the derived expression, under the assumption that all of its sub-expressions satisfy P.

- For conditional expressions, we assume that P holds on both sub-clauses.
- For production calls, we assume that P holds on all argument sub-expressions.
- For synthesized attribute accesses on a local attribute, we assume P holds on the local attribute's root production and its valid evaluable higher-order expressions, as well as the local's defining expression on its parent production.

For a non-circular grammar, these base and inductive cases suffice for a proof by induction on the structure of evaluable higher-order expressions.

C.2 Deriving New Local Trees from Parent Trees Using the Rewrite Rules

In any constant tree creation sequence, we can derive a pruned version of the generated tree from a pruned version of its predecessor (if it is not `INH`) via a non-empty rewrite sequence. Further, the last term in the rewrite sequence is not `INH`, which means the rewriting process can continue. The first term of the sequence is a pruned partially evaluated term in which all conditions in the local's defining expression have been evaluated, but in which attribute instance accesses have not been computed and are represented by the sub-trees on which they are to be evaluated. The first term is given by the function $eval_\alpha$ shown in Figure C.2. This function is defined to correspond to the definition in Figure 5.6 of the rules generated for each production. The rules model conditional expressions by generating rules for both clauses in each expression. **Lemma** 10 states that the first term is derivable from the tree term of the local attribute instance's defining node, via the one rule that corresponds to the actual evaluated values of the flags in each conditional expression. The rest of the sequence rewrites the pruned sub-trees of

$eval_\alpha : N \longrightarrow Expr \longrightarrow Attribution \longrightarrow RTerm$ returns the first term of the rewrite sequence that models the evaluation of a higher-order expression, in which all conditions have been evaluated, but in which attribute instance accesses are represented by the sub-trees on which they are evaluated.

For an expression e on a node n where $type_e(prod(n),\ e) \in (NT \cup T)$ and $\forall (n'\#a') \in dep(n,\ e,\ \Gamma)\ .\ \Gamma[n'\#a'] \neq \bot$, $eval_\alpha(n,\ e,\ \Gamma)$ is defined as follows:

$eval_\alpha(n,\ \#i,\ \Gamma) = term(child(n,i))$ (${i}^{\mathtt{th}}$ tree)
$eval_\alpha(n,\ \#0.a_I,\ \Gamma) = \mathtt{INH}$ (inherited attribute)
$eval_\alpha(n,\ \#i.a_S,\ \Gamma) = term(child(n,i))$ (${i}^{\mathtt{th}}$ child tree derives a_S)

$eval_\alpha(n,\ l.a_S,\ \Gamma) = eval_\alpha(n,\ e_L,\ \Gamma)$ (local's parent tree derives a_S)
where $(l = e_L) \in defs(prod(n))$

$eval_\alpha(n,\ q(e_1, ..., \ e_{n_q}),\ \Gamma)$ (evaluated tree term)
$= q(eval_\alpha(n,\ e_1,\ \Gamma), ...,\ eval_\alpha(n,\ e_{n_q},\ \Gamma))$

$eval_\alpha(n,\ c,\ \Gamma) = c$ (terminal symbol)

$eval_\alpha(n,\ \mathtt{if}\ e_C\ \mathtt{then}\ e_T\ \mathtt{else}\ e_E,\ \Gamma)$

$$= \begin{cases} eval_\alpha(n,\ e_T,\ \Gamma) & \text{if } eval(n,\ e_C,\ \Gamma) = \mathtt{true} \\ eval_\alpha(n,\ e_E,\ \Gamma) & \text{otherwise} \end{cases}$$ (evaluated sub-expression)

Figure C.2: The first term in the rewrite sequence that corresponds to the evaluation of a given evaluable higher-order expression.

this first term on which synthesized attribute instances are defined, to pruned versions of their evaluated values. The fact that these additional rewrites can be performed to generate the correct values can be shown inductively on the structure of the defining expressions. This is stated formally as **Lemma** 12.

Lemma 10: For any evaluable higher-order attribute instance $n\#a$ with defining expression e in a state $\langle \mathcal{T},\ \Gamma \rangle$ where $symbol(n) \approx K$, we can rewrite any u where $u \sqsubset_K term(n)$ to v where $v \sqsubset_K eval_\alpha(n,\ e,\ \Gamma))$ via a rule in $getRules(P)$. Further, if a's non-terminal type is of the same size as K, then $v \neq \texttt{INH}$.

In other words, if $u \sqsubset_K term(n)$ then $u \overset{\mathcal{R}}{\Longrightarrow} v$ where
$v \sqsubset_K eval_\alpha(n,\ e,\ \Gamma)$, $\mathcal{R} = getRules(P)$ and $v \neq \texttt{INH}$ if $type_a(a) \approx K$.

Proof: **Lemma** 10 is implied by **Lemma** 11.

Lemma 11: For any evaluable higher-order expression e, evaluated on a node n in a state $\langle \mathcal{T},\ \Gamma \rangle$ where $symbol(n) \approx K$, we can rewrite any u where $u \sqsubset_K term(n)$ to v where $v \sqsubset_K eval_\alpha(n,\ e,\ \Gamma)$, via at least one rewrite rule in $ruleRHSs(prod(n),\ e)$. Further, if e's non-terminal type is of the same size as K, then v is not $\texttt{INH}$.

Formally, at any state $\langle \mathcal{T},\ \Gamma \rangle$ and higher-order expression e where
- $t' \in \mathcal{T}$
- $n \in nodes(t')$
- $type_e(prod(n),\ e) \in NT\ \cup\ T$
- $\forall n'\#a' \in dep(n,\ e,\ \Gamma)\ .\ \Gamma[n'\#a'] \neq \bot$
- $symbol(n) \approx K$
- $u \sqsubset_K term(n)$

we have
- $u \overset{r}{\rightharpoondown} v$ where $v \sqsubset_K eval_\alpha(n,\ e,\ \Gamma)$ for some $r \in ruleRHSs(prod(n),\ e)$
- if $type_e(prod(n),\ e) \approx K$ then $v \neq \texttt{INH}$

Proof: Proof is by induction on the structure of e. Let $children(n) = n_1, ..., n_{n_p}$ and $p = prod(n)$. Since $symbol(n) \approx K$, we have $u = p(u_1, ..., u_{n_p})$ where $u_1 \sqsubset_K term(n_1), ..., u_{n_p} \sqsubset_K term(n_{n_p})$, by **Lemma** 5.

Base Cases:

- e is $\#i$:

- $u \overset{\#i}{\rightharpoondown} u_i$ where $u_i \sqsubset_K term(n_i) = eval_\alpha(n,\ \#i,\ \Gamma)$ and $ruleRHSs(p,\ \#i) = \{\#i\}$.

- If $symbol(n_i) \approx K$ then $u_i \neq \texttt{INH}$, by **Lemma** 5.

- e is c:

- $u \overset{c}{\rightharpoondown} c$ where $c \sqsubset_K c = eval_\alpha(n,\ c,\ \Gamma)$ and $ruleRHSs(p,\ c) = \{c\}$.
- $c \neq \texttt{INH}$.

- e is $\#i.a_S$:

- $u \overset{\#i}{\rightharpoondown} u_i$ where $u_i \sqsubset_K term(n_i) = eval_\alpha(n,\ \#i.a_S,\ \Gamma)$ and $ruleRHSs(p,\ \#i.a_S) = \{\#i\}$.
- If $type_a(a_S) \approx K$ then $symbol(n_i) \approx K$ and $u_i \neq \texttt{INH}$, by **Lemma** 5.

- e is $\#0.a_I$:

- $type_a(a_I) \succ K$.
- $u \overset{\texttt{INH}}{\rightharpoondown} \texttt{INH}$ where $\texttt{INH} \sqsubset_K term(n_i) = eval_\alpha(n,\ \#i.a_S,\ \Gamma)$ and $ruleRHSs(p,\ \#0.a_I) = \{\texttt{INH}\}$.

Inductive Cases:

- e is $l.a_S$:

- Assume
  - $(l = e_L) \in defs(p)$
  - $u \overset{r_L}{\rightharpoondown} v_L$ for some $v_L \sqsubset_K eval_\alpha(n,\ e_L,\ \Gamma)$ and $r_L \in ruleRHSs(p,\ e_L)$
- Since $eval_\alpha(n,\ l.a_S,\ \Gamma) = eval_\alpha(n,\ e_L,\ \Gamma)$ and $ruleRHSs(p,\ l.a_S) = ruleRHSs(p,\ e_L)$, we have $u \overset{r}{\rightharpoondown} v_L$ for some $v_L \sqsubset_K eval_\alpha(n,\ e,\ \Gamma)$ and $r \in ruleRHSs(p,\ e)$.
- If $type_a(a_S) \approx K$ then $type_e(p,\ e_L) \approx K$ and $v_L \neq \texttt{INH}$, by **Lemma** 5.

- e is **if** e_C **then** e_T **else** e_E:

- Assume
  - $u \overset{r_T}{\rightharpoondown} v_T$ for some $v_T \sqsubset_K eval_\alpha(n,\ e_T,\ \Gamma)$ and $r_T \in ruleRHSs(p,\ e_T)$
  - $u \overset{r_E}{\rightharpoondown} v_E$ for some $v_E \sqsubset_K eval_\alpha(n,\ e_E,\ \Gamma)$ and $r_E \in ruleRHSs(p,\ e_E)$
- If $eval(n,\ e_C,\ \Gamma) = \texttt{true}$ then $eval_\alpha(n,\ e,\ \Gamma) = eval_\alpha(n,\ e_T,\ \Gamma)$.
- If $eval(n,\ e_C,\ \Gamma) = \texttt{false}$ then $eval_\alpha(n,\ e,\ \Gamma) = eval_\alpha(n,\ e_E,\ \Gamma)$.
- Since $ruleRHSs(p,\ e) = ruleRHSs(p,\ e_T)\ \cup\ ruleRHSs(q,\ e_E)$, we have $u \overset{r}{\rightharpoondown} v$ for some $v \sqsubset_K eval_\alpha(n,\ e,\ \Gamma)$ and $r \in ruleRHSs(p,\ e)$.
- If $type_e(p,\ e_T) \approx type_e(p,\ e_E) \approx K$ then $v_T \neq \texttt{INH}$ and $v_E \neq \texttt{INH}$ by **Lemma** 5, and so $v \neq \texttt{INH}$.

- e is $q(e_1, ...,\ e_{n_q})$

- Assume
  - $u \xrightarrow{r_1} v_1$ for some $v_1 \sqsubset_K eval_\alpha(n,\ e_1,\ \Gamma)$ and $r_1 \in ruleRHSs(p,\ e_1)$
  - ...
  - $u \xrightarrow{r_{n_q}} v_{n_q}$ for some $v_{n_q} \sqsubset_K eval_\alpha(n,\ e_{n_q},\ \Gamma)$ and $r_{n_q} \in ruleRHSs(p,\ e_{n_q})$
- Let v be $q(v_1, ..., v_{n_q})$.
- $v \sqsubset_K q(eval_\alpha(n,\ e_1,\ \Gamma), ...,\ eval_\alpha(n,\ e_{n_q},\ \Gamma))$, by the definition of $\sqsubset_K$.
- So we have $u \xrightarrow{q(r_1,...,r_{n_q})} v$ for some $v \sqsubset_K q(eval_\alpha(n,\ e_1,\ \Gamma), ...,\ eval_\alpha(n,\ e_{n_q},\ \Gamma))$ and $r_1 \in ruleRHSs(p,\ e_1), ..., r_{n_q} \in ruleRHSs(p,\ e_{n_q})$.
- $eval_\alpha(n,\ q(e_1, ...,\ e_{n_q}),\ \Gamma) = q(eval_\alpha(n,\ e_1,\ \Gamma), ...,\ eval_\alpha(n,\ e_{n_q},\ \Gamma))$.
- $ruleRHSs(p,\ q(e_1, ...,\ e_{n_q})) = \{\ q(r_1, ..., r_{n_q}) \mid r_i \in ruleRHSs(p,\ e_i), 1 \leq i \leq n_q\ \}$.
- Therefore, $u \xrightarrow{r} v$ for some $v \sqsubset_K eval_\alpha(n,\ e,\ \Gamma)$ and $r \in ruleRHSs(p,\ e)$.
- As $v \neq \texttt{INH}$, the second condition in the lemma is trivially satisfied.

Lemma 12: For any evaluable higher-order attribute instance $n\#a$ with defining expression e in a state $\langle \mathcal{T},\ \Gamma \rangle$ where $symbol(n) \approx K$, we can rewrite any u where $u \sqsubset_K eval_\alpha(n,\ e,\ \Gamma)$ to v where $v \sqsubset_K \Gamma[n\#a]$, using the rules in $getRules(P)$. Further, if a's non-terminal type is of the same size as K, then v is not $\texttt{INH}$.

In other words, if $u \sqsubset_K eval_\alpha(n,\ e,\ \Gamma)$ then $u \overset{\mathcal{R}}{\Longrightarrow}{}^{*}\ v$ where $v \sqsubset_K \Gamma[n\#a]$, $\mathcal{R} = getRules(P)$ and $v \neq \texttt{INH}$ if $type_a(a) \approx K$.

Proof: **Lemma** 12 is implied by **Lemma** 13.

Lemma 13: For any evaluable higher-order expression e, evaluated on a node n in a state $\langle \mathcal{T},\ \Gamma \rangle$ where $symbol(n) \approx K$, we can rewrite any u where $u \sqsubset_K eval_\alpha(n,\ e,\ \Gamma)$ to v where $v \sqsubset_K eval(n,\ e,\ \Gamma)$, using the rules in $getRules(P)$, if the evaluated values of e's required attribute instances are similarly derivable from their nodes' sub-trees. Further, if e's non-terminal type is of the same size as K, then v is not $\texttt{INH}$.

Formally, at any state $\langle \mathcal{T},\ \Gamma \rangle$ and higher-order expression e where
- $t' \in \mathcal{T}$
- $n \in nodes(t')$
- $type_e(prod(n),\ e) \in NT\ \cup\ T$
- $symbol(n) \approx type_e(prod(n), e) \approx K$ for some $K \in NT$
- $u \sqsubset_K eval_\alpha(n,\ e,\ \Gamma)$
- $\forall n'\#a' \in dep(n,\ e,\ \Gamma)$. $(\Gamma[n'\#a'] \neq \bot$ and
  - for any $u' \sqsubset_K term(n')$ we have $u' \overset{\mathcal{R}}{\Longrightarrow}{}^{+}\ v'$ for some $v' \sqsubset_K \Gamma[n'\#a']$ where $\mathcal{R} = getRules(P)$

- if $type_a(a_S) \approx K$ then $v' \neq \texttt{INH}$).

we have

- $u \overset{\mathcal{R}}{\Longrightarrow}{}^{*} v$ where $v \sqsubset_K eval(n,\ e,\ \Gamma)$, $\mathcal{R} = getRules(P)$
- if $type_e(prod(n), e) \approx K$ then $v \neq \texttt{INH}$

Proof: Proof is by induction on the structure of e. Let $children(n) = n_1, ..., n_{n_p}$, $p = prod(n)$ and $\mathcal{R} = getRules(P)$.

Base Cases:

- e is $\#i$:

- Since $u \sqsubset_K eval_\alpha(n,\ \#i,\ \Gamma) = term(n_i) = eval(n,\ \#i,\ \Gamma)$, we trivially have $u \overset{\mathcal{R}}{\Longrightarrow}{}^{*} v$ where $v \sqsubset_K eval(n,\ \#i,\ \Gamma)$.
- If $symbol(n_i) \approx K$ then $u \neq \texttt{INH}$ by **Lemma** 5, and $v \neq \texttt{INH}$.

- e is c:

- Since $u \sqsubset_K eval_\alpha(n,\ c,\ \Gamma) = c = eval(n,\ c,\ \Gamma)$, we trivially have $u \overset{\mathcal{R}}{\Longrightarrow}{}^{*} v$ where $v \sqsubset_K eval(n,\ c,\ \Gamma)$.
- $c \neq \texttt{INH}$.

- e is $\#i.a_S$:

- If $u = \texttt{INH}$, we have $\texttt{INH} \sqsubset_K eval(n,\ \#i.a_S,\ \Gamma)$, and so $u \overset{\mathcal{R}}{\Longrightarrow}{}^{*} v$ for some $v \sqsubset_K eval(n,\ \#i.a_S,\ \Gamma)$.
- Assume $u \neq \texttt{INH}$.
- Since $n_i\#a_S \in dep(n,\ e,\ \Gamma)$, for any $u_i \sqsubset_K term(n_i)$ we have
  - $u_i \overset{\mathcal{R}}{\Longrightarrow}{}^{+} v_i$ for some $v_i \sqsubset_K \Gamma[n_i\#a_S]$
  - if $type_a(a_S) \approx K$ then $v_i \neq \texttt{INH}$
- $eval_\alpha(n,\ \#i.a_S,\ \Gamma)) = term(n_i)$.
- $eval(n,\ \#i.a_S,\ \Gamma) = \Gamma[n_i\#a_S]$.
- So we have $u \overset{\mathcal{R}}{\Longrightarrow}{}^{*} v$ for some $v \sqsubset_K eval(n,\ \#i.a_S,\ \Gamma)$.
- If $type_a(a_S) \approx K$ we have $v \neq \texttt{INH}$.

- e is $\#0.a_I$:

- $eval_\alpha(n,\ \#0.a_I,\ \Gamma) = \texttt{INH}$ and $\texttt{INH} \sqsubset_K eval(n,\ \#0.a_I,\ \Gamma)$.
- So for any $u \sqsubset_K eval_\alpha(n,\ \#0.a_I,\ \Gamma)$, we have $u \overset{\mathcal{R}}{\Longrightarrow}{}^{*} v$ for some $v \sqsubset_K eval(n,\ \#0.a_I,\ \Gamma)$.
- $type_a(a_I) \succ K$ and so the second condition in the lemma is trivially satisfied.

Inductive Cases:

- e is $l.a_S$:

- If $u = \texttt{INH}$, we have $\texttt{INH} \sqsubset_K \mathit{eval}(n,\ l.a_S,\ \Gamma)$, and so we have $u \stackrel{\mathcal{R}}{\Longrightarrow}^* v$ for some $v \sqsubset_K \mathit{eval}(n,\ l.a_S,\ \Gamma)$.
- Assume $u \neq \texttt{INH}$.
- Assume
  - $(l = e_L) \in \mathit{defs}(p)$
  - for any $u_L \sqsubset_K \mathit{eval}_\alpha(n,\ e_L,\ \Gamma)$ we have
    - $u_L \stackrel{\mathcal{R}}{\Longrightarrow}^* v_L$ for some $v_L \sqsubset_K \mathit{eval}(n,\ e_L,\ \Gamma)$
    - if $\mathit{type}_a(l) \approx K$ then $v_L \neq \texttt{INH}$
- Since $(\Gamma[n\#l])\#a_S \in \mathit{dep}(n,\ e,\ \Gamma)$, for any $u_L \sqsubset_K \mathit{term}(\Gamma[n\#l])$ we have
  - $u_L \stackrel{\mathcal{R}}{\Longrightarrow}^+ v$ for some $v \sqsubset_K \Gamma[(\Gamma[n\#l])\#a_S]$
  - if $\mathit{type}_a(a_S) \approx K$ then $v \neq \texttt{INH}$
- $\mathit{eval}_\alpha(n,\ l.a_S,\ \Gamma) = \mathit{eval}_\alpha(n,\ e_L,\ \Gamma)$.
- $\mathit{eval}(n,\ e_L,\ \Gamma) = \mathit{term}(\Gamma[n\#l])$.
- $\mathit{eval}(n,\ l.a_S,\ \Gamma) = \Gamma[(\Gamma[n\#l])\#a_S]$.
- So we have $u \stackrel{\mathcal{R}}{\Longrightarrow}^* v$ for some $v \sqsubset_K \mathit{eval}(n,\ l.a_S,\ \Gamma)$.
- If $\mathit{type}_a(a_S) \approx K$, then $v \neq \texttt{INH}$.

- e is $\texttt{if}\ e_C\ \texttt{then}\ e_T\ \texttt{else}\ e_E$:

- If $u = \texttt{INH}$, we have $\texttt{INH} \sqsubset_K \mathit{eval}(n,\ \texttt{if}\ e_C\ \texttt{then}\ e_T\ \texttt{else}\ e_E,\ \Gamma)$, and so $u \stackrel{\mathcal{R}}{\Longrightarrow}^* v$ for some $v \sqsubset_K \mathit{eval}(n,\ \texttt{if}\ e_C\ \texttt{then}\ e_T\ \texttt{else}\ e_E,\ \Gamma)$.
- Assume $u \neq \texttt{INH}$.
- We have the following inductive assumptions:
  - if $u_T \sqsubset_K \mathit{eval}_\alpha(n,\ e_T,\ \Gamma)$, $u_T \stackrel{\mathcal{R}}{\Longrightarrow}^* v_T$ where $v_T \sqsubset_K \mathit{eval}(n,\ e_T,\ \Gamma)$
  - if $u_E \sqsubset_K \mathit{eval}_\alpha(n,\ e_E,\ \Gamma)$, $u_E \stackrel{\mathcal{R}}{\Longrightarrow}^* v_E$ where $v_E \sqsubset_K \mathit{eval}(n,\ e_E,\ \Gamma)$
- If $\mathit{eval}(n,\ e_C,\ \Gamma) = \texttt{true}$, then
  - $\mathit{eval}_\alpha(n,\ e,\ \Gamma) = \mathit{eval}_\alpha(n,\ e_T,\ \Gamma)$
  - $\mathit{eval}(n,\ e,\ \Gamma) = \mathit{eval}(n,\ e_T,\ \Gamma)$
- If $\mathit{eval}(n,\ e_C,\ \Gamma) = \texttt{false}$, then
  - $\mathit{eval}_\alpha(n,\ e,\ \Gamma) = \mathit{eval}_\alpha(n,\ e_E,\ \Gamma)$
  - $\mathit{eval}(n,\ e,\ \Gamma) = \mathit{eval}(n,\ e_E,\ \Gamma)$
- In either case, $u \stackrel{\mathcal{R}}{\Longrightarrow}^* v$ for some $v \sqsubset_K \mathit{eval}(n,\ e,\ \Gamma)$.
- If $\mathit{symbol}(n) \approx \mathit{type}_e(p,\ e_T) \approx \mathit{type}_e(p,\ e_E) \approx K$, then by the inductive assumption, $v_T \neq \texttt{INH}$ and $v_E \neq \texttt{INH}$, and so $v \neq \texttt{INH}$.

- e is $q(e_1, ...,\ e_{n_q})$:

- $u = p(u_1, ..., u_{n_q})$.
- Assume for $1 \leq i \leq n_q$ we have
  - for any $u_i \sqsubset_K eval_\alpha(n,\ e_i,\ \Gamma)$ we have $u_i \overset{\mathcal{R}}{\Longrightarrow}^* v_i$ for some $v_i \sqsubset_K eval(n,\ e_i,\ \Gamma)$
- Let v be $q(v_1, ..., v_{n_q})$.
- $eval(n,\ e,\ \Gamma) = q(eval(n,\ e_1,\ \Gamma), ..., eval(n,\ e_{n_q},\ \Gamma))$.
- We therefore have $u \overset{\mathcal{R}}{\Longrightarrow}^* v$ for some $v \sqsubset_K eval(n,\ e,\ \Gamma)$.
- $v \neq \texttt{INH}$, and so the second condition of the lemma is trivially satisfied.

Lemma 6: For any evaluable higher-order attribute instance $n\#a$ in a state $\langle \mathcal{T},\ \Gamma \rangle$ with defining expression e where $symbol(n) \approx K$ for some $K \in NT$, we can rewrite any u where $u \sqsubset_K term(n)$ to v where $v \sqsubset_K \Gamma[n\#a]$, via a non-empty rewrite sequence of rules in $getRules(P)$. Further, if a's non-terminal type is of the same size as K, then v is not $\texttt{INH}$.

Formally, at any state $\langle \mathcal{T},\ \Gamma \rangle$ where
- $t' \in \mathcal{T}$
- $n\#a \in instances(t')$ with defining expression e
- $type_a(a) \in NT\ \cup\ T$
- $\forall n'\#a' \in dep(n,\ e,\ \Gamma)\ .\ \Gamma[n'\#a'] \neq \bot$
- $symbol(n) \approx K \in NT$
- $u \sqsubset_K term(n)$

we have
- $u \overset{\mathcal{R}}{\Longrightarrow}^+ v$ where $v \sqsubset_K \Gamma[n\#a]$ and $\mathcal{R} = getRules(P)$
- if $type_a(a) \approx K$ then $v \neq \texttt{INH}$

Proof:

- Let $\mathcal{R} = getRules(P)$.
- By **Lemma** 10, $u \xrightarrow{r} u'$ for some $u' \sqsubset_K eval_\alpha(n,\ e,\ \Gamma)$ and $r \in ruleRHSs(p,\ e)$.
- By the definition of $getRules$, $\forall r \in ruleRHSs(p,\ e)\ .\ p(\#1, ..., \#n_p) \longrightarrow r \in getRules(P)$, since $type_a(a) \in NT\ \cup\ T$.
- Therefore, we have $u \overset{\mathcal{R}}{\Longrightarrow} u'$.
- By **Lemma** 10, if $type_a(a) \approx K$ then $u' \neq \texttt{INH}$.
- We can show inductively that $u' \overset{\mathcal{R}}{\Longrightarrow}^* v$ for some $v \sqsubset_K eval(n,\ e,\ \Gamma)$ such that $v \neq \texttt{INH}$ if $type_a(a) \approx K$.
- Therefore we have $u \overset{\mathcal{R}}{\Longrightarrow}^+ v$ for some $v \sqsubset_K eval(n,\ e,\ \Gamma)$ such that $v \neq \texttt{INH}$ if $type_a(a) \approx K$.

Induction is on the number of higher-order attribute instances evaluated.

Base Case:

- For the first higher-order attribute instance evaluated, we have $dep(n,\ e,\ \Gamma) = \{\ \}$.
- We therefore trivially have
  $\forall n'\#a' \in dep(n,\ e,\ \Gamma)\ .\ (\Gamma[n'\#a'] \neq \perp$ and
  - for any $u' \sqsubset_K term(n')$ we have $u' \stackrel{\mathcal{R}}{\Longrightarrow}^{+} v'$ for some $v' \sqsubset_K \Gamma[n'\#a']$
  - if $type_a(a') \approx K$ then $v' \neq \texttt{INH}$).
- By **Lemma** 12, $u' \stackrel{\mathcal{R}}{\Longrightarrow}^{*} v$ for some $v \sqsubset_K eval(n,\ e,\ \Gamma)$ such that $v \neq \texttt{INH}$ if $type_a(a) \approx K$.

Inductive Case:

- For a non-initial evaluable higher-order attribute instance, all required attribute instances are higher-order, as described in Section C.1.
- Therefore by the inductive assumption
  $\forall n'\#a' \in dep(n,\ e,\ \Gamma)\ .\ (\Gamma[n'\#a'] \neq \perp$ and
  - for any $u' \sqsubset_K term(n')$ we have $u' \stackrel{\mathcal{R}}{\Longrightarrow}^{+} v'$ for some $v' \sqsubset_K \Gamma[n'\#a']$
  - if $type_a(a') \approx K$ then $v' \neq \texttt{INH}$).
- By **Lemma** 12, $u' \stackrel{\mathcal{R}}{\Longrightarrow}^{*} v$ for some $v \sqsubset_K eval(n,\ e,\ \Gamma)$ such that $v \neq \texttt{INH}$ if $type_a(a) \approx K$.