Extensible Languages via Modular Declarative Specifications

A THESIS
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

May, 2012
Extensible Languages via Modular Declarative Specifications

by Lijesh Krishnan Manjacheri

ABSTRACT

Silver is an extensible language framework developed to address the challenge of extending programming languages with high-level abstractions to serve the needs of programmers in specific domains. To make the process of language extension more modular, Silver separates out the stages of host language modeling, extension development, and extension composition. The syntax and semantics of the host language are defined by in an extensible host language specification. Domain-specific language features with new syntax and semantic analyses are independently developed by extension developers with domain expertise, and specified declaratively via self-contained extension specifications. Given a host specification and a set of extension specifications specified by a user programmer, the Silver tool constructs a generated compiler for an extended version of a host language.

The problem of specifying and composing host and extension semantics (such as pretty-printing, error checking and source-level transformations) is handled via the evaluation by Silver's higher-order attribute grammar system. The Silver approach stresses the importance of specification analyses that detect composition errors, particularly those that can be performed at extension development time. An example of such an analysis is a check for termination of tree creation during higher-order attribute evaluation. Combined with the circularity test, it provides the extension writer or user with a guarantee that the resulting compiler will not fail to terminate on account of improper attribute evaluation.

The dissertation makes two contributions toward exploring the use of a declarative, attribute grammar-based tool (extended with features such as forwarding, aspects, higher-order attributes and collections) for writing modular, composable and statically analyzable language specifications. First, we give a demonstration of the feasibility of the library model of extensibility by describing two examples of using Silver to write host and extension specifications for real-world languages. We describe ableJ, an extensible
host specification for Java 1.4 that generates front-end translators from extended code to valid Java 1.4 code. We describe the challenges involved in designing and writing the host specification so that its type system and environment can be conveniently accessed and modified by extensions. Extensions written to ableJ range from a simple macro-like enhanced for extension, to more complex ones for complex number types, auto-boxing and algebraic data-types with pattern-matching. We briefly describe a second example in which the Silver specification language is extended with constructs to specify and perform data-flow analyses.

We also present a conservative analysis that checks HOAG specifications for infinite tree creation during attribution. It does this via a two-part check that first attempts to construct a terminating set of rewrite rules to model attribution in the absence of inherited attributes, and then restricts the number of inherited accesses by ordering the grammar’s non-terminals based on how they decorate each other as inherited attributes. We have run the analysis on the ableJ host and its extension specifications to demonstrate that the analysis is powerful enough to show termination of non-trivial grammars.
Acknowledgements

I thank everyone.
Dedication

To Dad.
Contents

Abstract i
Acknowledgements ii
Dedication iii
List of Tables ix
List of Figures x

1 A Library Model for Language Extensions 1
  1.1 Bridging the Programming Semantic Gap . . . . . . . . . . . . . . . . . 1
  1.2 A Modular Approach to Language Extensions Similar to Libraries . . 2
    1.2.1 An Example of the Library Model in Action . . . . . . . . . . . . 4
  1.3 Challenges to Implementing a Modular Extensible Language Framework 7
    1.3.1 Issues with Specifying Concrete Syntax in an Extensible Fashion 7
    1.3.2 Issues with Specifying Semantics in an Extensible Fashion . . . 8
  1.4 Silver: An Extensible Language Framework . . . . . . . . . . . . . . . 8
    1.4.1 Silver Specifications . . . . . . . . . . . . . . . . . . . . . . . . . . 9
    1.4.2 Extensible Concrete Syntax via Copper's Context-Aware Scanning 10
    1.4.3 Extensible Semantics via Higher-Order Attribute Grammars . . . 11
    1.4.4 Static Analysis of Silver Specifications for Composability . . . 12
  1.5 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
    1.5.1 Extensible Silver Specifications for Real-World Languages . . . 14
    1.5.2 A Static Analysis for HOAG Termination . . . . . . . . . . . . . . 15
1.6 Dissertation Outline ................................................. 15

2 An Extensible Specification for Java .................................. 17
  2.1 Introduction ....................................................... 17
    2.1.1 Bridging the Programming Semantic Gap ...................... 18
    2.1.2 Writing Composable Extensions in Silver .................... 18
    2.1.3 ableJ: A Silver Tool to Add Features to Java ............... 19
    2.1.4 Reusing Host Semantics to Write Modular Extensions ....... 20
    2.1.5 Chapter Outline ............................................. 21
  2.2 Sample Language Extensions ...................................... 22
    2.2.1 Adding an Enhanced \texttt{for} Loop ......................... 23
    2.2.2 Adding Complex Numbers as a Primitive Type ............... 24
    2.2.3 Adding Auto-Boxing and Auto-Unboxing ..................... 25
    2.2.4 Embedding the SQL Query Language .......................... 26
    2.2.5 Adding Algebraic Data-types and Pattern-Matching ......... 27
  2.3 The Host Language Specification .................................. 29
    2.3.1 Specifying the Java 1.4 Concrete Syntax .................... 31
    2.3.2 Specifying the Java 1.4 Semantics Using Higher-Order Attribute
         Grammars ................................................... 33
  2.4 Adding New Constructs ............................................ 40
    2.4.1 Adding the Extended \texttt{for} Loop ......................... 40
    2.4.2 Adding the Complex Number Type ............................ 42
  2.5 Composing the Host and Extensions ................................ 44
  2.6 Accessing and Adding to the Host Environment ................... 45
    2.6.1 \texttt{EnvItem}: A Data Structure to Store Declaration Information .. 45
    2.6.2 The Scoped Environment as a List of Lists .................. 46
    2.6.3 Variable Look-Ups .......................................... 48
  2.7 Accessing and Extending the Host Type System .................... 49
    2.7.1 Types and \texttt{TypeReps} .................................. 51
    2.7.2 Specifying Sub-Types and Mechanisms for Run-Time Conversion 53
    2.7.3 Operator Overloading ........................................ 57
  2.8 Discussion ...................................................... 59
# Extending Silver for Data-Flow Analysis

## 3.1 Introduction

### 3.1.1 Chapter Outline

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2 Specifying the Data-Flow Side-Conditions of Program Transformations</td>
<td>62</td>
</tr>
<tr>
<td>as Temporal Logic Formulas That Can Be Model-Checked on the CFG</td>
<td></td>
</tr>
<tr>
<td>3.3 Constructing the CFG Nodes and Labels from an AST in an AG Framework</td>
<td>65</td>
</tr>
<tr>
<td>3.4 A Silver Extension for Specifying and Executing Data-Flow Conditions</td>
<td>67</td>
</tr>
<tr>
<td>3.4.1 Specifying the Nodes, Edges and Labels of the Control-Flow Graph</td>
<td>69</td>
</tr>
<tr>
<td>3.4.2 Model-Checking CTL Formulas with NuSMV to Generate Optimized Code</td>
<td>71</td>
</tr>
</tbody>
</table>

## 3.5 Discussion

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5 Discussion</td>
<td>73</td>
</tr>
</tbody>
</table>

# Abstracting Higher-Order Attribution with Term Rewrite Rules

## 4.1 Simpler Abstractions to Model Higher-Order Attribution

### 4.1.1 Chapter Outline

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2 A Restricted Class of Attribute Grammars and a Simple Attribute Ev-</td>
<td>79</td>
</tr>
<tr>
<td>aluation Model</td>
<td></td>
</tr>
<tr>
<td>4.2.1 Restrictions on Attribute Definitions and Expressions</td>
<td>79</td>
</tr>
<tr>
<td>4.2.2 The Evaluation Model</td>
<td>86</td>
</tr>
<tr>
<td>4.3 Reducing the HOAG Termination Problem to Disproving the Existence</td>
<td>96</td>
</tr>
<tr>
<td>of Infinite Tree Creation Sequences</td>
<td></td>
</tr>
<tr>
<td>4.4 Rewrite Rule Sequences as a Simple Abstraction of Tree Creation Se-</td>
<td>97</td>
</tr>
<tr>
<td>quences in the Absence of Inherited Attributes</td>
<td></td>
</tr>
<tr>
<td>4.4.1 Generating Rewrite Rules from Higher-Order Attribute Definitions</td>
<td>101</td>
</tr>
<tr>
<td>4.4.2 Constructing a Rewrite Sequence Corresponding to a Tree Creation</td>
<td>102</td>
</tr>
<tr>
<td>Sequence</td>
<td></td>
</tr>
<tr>
<td>4.4.3 The Rules are Useful and Correct, Albeit Conservative</td>
<td>104</td>
</tr>
<tr>
<td>4.5 Ordering Trees to Limit Inherited Accesses in Tree Sequences</td>
<td>106</td>
</tr>
<tr>
<td>4.5.1 The Rules Do Not Work for Inherited Attributes</td>
<td>106</td>
</tr>
<tr>
<td>4.5.2 Bounding the Number of Inherited Accesses in Tree Creation Sequ-</td>
<td>110</td>
</tr>
<tr>
<td>ences</td>
<td></td>
</tr>
<tr>
<td>4.5.3 Constant Tree Creation Sequences Can Be Modeled by the Rules</td>
<td>114</td>
</tr>
</tbody>
</table>
4.5.4 A Procedure to Construct the Desired Ordering to Limit Inherited Accesses ........................................... 115

5 Related Work and Conclusion .................................. 121
5.1 Related Work ...................................................... 121

Appendix A. Technical Material from Chapter 4 (The Termination Analysis) ................................................. 123
A.1 Introduction and Preliminary Definitions ......................... 123
A.1.1 Chapter Outline .................................................. 123
A.1.2 Context-FreeGrammars and Attribute Grammars .................. 125
A.1.3 Syntax Nodes and Trees ........................................... 127
A.1.4 Tree Terms .......................................................... 127
A.1.5 Attribute Instances and Attributions ............................... 128
A.1.6 Evaluation States and Sequences .................................. 129
A.1.7 Notational Conventions ............................................. 130
A.2 Section 4.2 (A Restricted Class of Attribute Grammars) .......... 131
A.2.1 Attribute Definitions and Expressions ............................ 131
A.2.2 Type-Correct Definitions and Expressions .......................... 131
A.2.3 Defining the Steps in an Evaluation Sequence ..................... 134
A.3 Section 4.3 (Reducing the Problem to Disproving Infinite Tree Creation Sequences) ............................... 136
A.3.1 Reducing Infinite Evaluation to Infinite Tree Creation ............. 136
A.3.2 Higher-Order Tree Creating Expressions .......................... 142
A.4 Section 4.4 (Modeling Tree Creation During Evaluation with Rewrite Rules) .............................. 145
A.4.1 Constructing Rewrite Rules to Model Tree Creation ............... 145
A.4.2 Deriving Evaluated Local Trees from their Parent Sub-Trees Using the Rules ........................................ 147
A.4.3 Constructing a Rewriting Sequence for a Tree Creation Sequence 155
A.5 Section 4.5 (Ordering Trees to Limit Accesses to Inherited Attributes) ............................................. 157
A.5.1 Ordering Non-Terminals to Restrict Inherited Access During Tree Creation ........................................ 157
A.5.2 Ordering Non-Terminals to Restrict Inherited Access During Tree
Creation .......................................................... 158
A.5.3 For an Infinite Tree Creation Sequence, There is an Infinite Con-
stant Tree Creation Sequence ................................. 159
A.5.4 The Rules Do Model Constant Tree Creation Steps ............. 160
A.5.5 For Every Constant Tree Creation Sequence, There is a Rewrite
Sequence of the Same Length ............................... 169
List of Tables

3.1 The temporal operators of the Computational Tree Logic. 64
List of Figures

1.1 Code written in an extended version of Java 1.4, and equivalent Java 1.4 code. ........................................ 4
1.2 An overview of the actors, specifications and code in the library model of extensibility. ........................................ 5
1.3 Overview of the declarations in a Silver specifications. ................. 9
2.1 Code written in an extended version of Java 1.4, and equivalent Java 1.4 code. ........................................ 20
2.2 Using the enhanced for construct to iterate over arrays and collections, and the equivalent Java 1.4 code that is generated. ................. 23
2.3 Code in a version of Java in which complex numbers are a primitive type, and the equivalent Java 1.4 code that is generated. ................. 24
2.4 Code in a version of Java 1.4 with auto-boxing and auto-unboxing, and the equivalent Java 1.4 code that is generated. ................. 25
2.5 Code written in a version of Java embedded with the SQL query language, and the equivalent Java 1.4 code that is generated. ................. 26
2.6 Haskell code defining an algebraic data-type for lists. ................. 27
2.7 Functional-style code defining an algebraic data-type for lists. ................. 28
2.8 The equivalent Java 1.4 code that is generated for the extended code in Figure 2.7. ........................................ 30
2.9 Specifying Java 1.4’s lexicon in Silver. ........................................ 32
2.10 Specifying Java 1.4’s concrete syntax in Silver. ........................................ 34
2.11 Host specification declarations in ableJ. ........................................ 37
2.12 Adding an enhanced for loop to Java 1.4. See Figure 2.2 for examples of generated code. ........................................ 41
2.13 Adding complex numbers as a primitive type to Java 1.4 .......................... 43
2.14 Composing the Java 1.4 host with the SQL and algebraic data-type extensions. ................................................. 44
2.15 Constructing and gathering up environment items at the point of declaration. .......................................................... 46
2.16 Constructing and passing scoped environments down the tree. .......... 47
2.17 Adding to the environment in the enhanced for loop ....................... 47
2.18 Looking up variables in the environment ........................................ 48
2.19 Type representations in ableJ ...................................................... 50
2.20 Adding a new type for complex numbers ....................................... 51
2.21 Type-checking in ableJ ............................................................... 52
2.22 Type-checking in the enhanced for extension .................................. 53
2.23 Specifying sub-type relations ....................................................... 54
2.24 Specifying new sub-type relations with the complex type ................ 55
2.25 Using collection attributes to define specialized assignment trees ...... 56
2.26 Use of run-time conversion in the host assignment statement .......... 56
2.27 Using collection attributes to define specialized assignment trees ...... 58
2.28 Performing auto-boxing via collection attributes .......................... 59
3.1 Performing dead-code elimination on a C‐ program ......................... 63
3.2 The control flow graph of the program in Figure 3.1 ......................... 63
3.3 The abstract syntax tree of the program in Figure 3.1 ....................... 65
3.4 Defining the CFG on the AST for the program in Figure 3.1 ............. 66
3.5 Specifying the labels on the nodes of the control-flow graph ................ 68
3.6 Specifying the construction of the control-flow graph for C‐ programs. .. 70
3.7 Constructing the NuSMV model to perform dead code elimination ....... 72
4.1 An informal description of the process of attribute evaluation process for higher-order attribute grammars .......................... 75
4.2 $G_0$: a complete, non-circular grammar for which tree creation does not terminate .................................................. 76
4.3 A decision procedure that statically analyses a grammar for termination of tree creation .............................................. 78
4.4 $G_1$: a grammar for an imperative language without inherited attributes. 82
4.5 \( G_2 \): a grammar with inherited attributes decorating non-terminals that are inherited attributes, part 1 of 3. 83
4.6 \( G_2 \): a grammar with inherited attributes decorating non-terminals that are inherited attributes, part 2 of 3. 84
4.7 \( G_2 \): a grammar with inherited attributes decorating non-terminals that are inherited attributes, part 3 of 3. 85
4.8 \( G_3 \): a grammar for which tree creation does not terminate, part 1 of 2. 105
4.9 \( G_3 \): a grammar for which tree creation does not terminate, part 2 of 2. 106
4.10 \( G_4 \): a grammar whose non-terminals cannot be ordered as described in
Section 4.5.2. 112
4.11 Algorithm to assign non-terminals values that satisfy the conditions in
Section 4.5.2, part 1 of 2. 116
4.12 Algorithm to assign non-terminals values that satisfy the conditions in
Section 4.5.2, part 2 of 2. 117
4.13 The phases of the analysis when it is run on the grammar \( G_2 \) defined in
Figures 4.5, 4.6 and 4.7. 119
4.14 The phases of the analysis when it is run on the grammar \( G_4 \) defined in
Figure 4.10. 120
A.1 The syntax of attribute grammar definitions and expressions. 132
A.2 An example of the definition of an attribute grammar. 133
A.3 Defining the type of attribute expressions. 134
A.4 Valid attribute definition expressions. 135
A.5 Valid attribute definitions. 136
A.6 Defining the non-initial states in an evaluation sequence. 137
A.7 Defining the set of attribute instances required to evaluate an expression. 138
A.8 Defining the evaluated value of an attribute definition expression. 139
A.9 The set of expressions that are evaluated to local tree values, and their
required instances. 143
A.10 Defining the right-hand sides of the rewrite rules generated to model a
given higher-order expression. 146
A.11 The first term in an evaluated attribute instance’s rewrite sequence. 148
A.12 Extending the definitions of rewrite rules and terms to handle grammars with higher-order inherited attributes. .......................... 157
Chapter 1

A Library Model for Language Extensions

1.1 Bridging the Programming Semantic Gap

The question of extending programming languages to serve the needs of programmers in specific domains is an interesting and challenging one. Ideally, programmers could make use of domain specific knowledge and program using constructs that implement abstractions in the domain. This would result in better and easier to maintain code. However, programmers must overcome the semantic distance between any high-level domain-specific abstractions that they have knowledge of, or expertise in, and the non-domain specific general features available for programming in most programming languages.

Programming language do provide facilities for developing new abstractions, such as higher-order functions, classes, generics and parametric polymorphism. These abstractions may implement the desired functionality of the abstraction but do not provide new language constructs with new and more suitable syntax.

Domain-specific languages (DSL), on the other hand, 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 reduces the semantic gap between the general programming language and the specific needs of the program. Examples of domain-specific languages are given in [1].
Another advantage of DSLs is that unlike general purpose languages, they can perform domain-specific optimizations and other analysis. The drawback of the DSL approach is that it is often not economically viable to construct entire new languages and compilers, with associated tools if the programmer base is not large. And even if DSLs are available, it is often the case that a given programming problem spans more than one domain, compounding the viability issue. And developing full-fledged compilers for small groups of programmers does not make economic sense [2]. The problem is compounded if the programmer’s needs span multiple domains.

It is unlikely that existing programming languages provide all the features (whether general purpose or domain specific) that might be needed to solve a given programming problem. Programmers are thus forced to write their solutions in a form restricted by the idioms supported by the general purpose language. Using low-level representations is cumbersome, prone to errors and not easy to maintain.

1.2 A Modular Approach to Language Extensions Similar to Libraries

Ideally, we want extending languages to be as easy as importing libraries. A user with no implementation knowledge can import independently written specifications into a host, and expect that in most cases these extensions would work together.

The library model of language extensibility separates out the stages of host language modeling, extension development, and extension composition to make the process of language extension more modular. Thus it separates out the three roles of host language specification writer, extension writer and user programmer, who have varying degrees of compiler expertise, domain specific knowledge and programming needs. It allows for communication, division of labor and information exchange between these disparate actors to solve a situation that traditional compiler development does not address. Unlike the host language and extension writers, who must possess a fair amount of programming language design and implementation knowledge, the user programmer would be spared by the modular nature of the model from having to deal with low-level details of the interactions between the new constructs and the host language compiler. We now look at these three stages in more detail.
The Host Language Specification  The syntax and semantics of the host language are defined in a host language specification. The host specification writer must make important choices while designing the host based on how it is likely to be extended by extension writers. He must think about what aspects of the host language and the compilation process extension writers would likely want to access. Aspects such as the host concrete syntax, type system and environment should be implemented in an extensible way to allow for new constructs and new analyses to be added. Access should be made available through an interface that allows for modular extensions, but also provides some measure of safety so that extensions are likely to compose. Modularity and composability are more likely if the host semantics is specified using a high-level declarative framework. For example, the host could define abstractions to compute lists of program errors to which extensions could make their own contributions.

Extension Specifications  Extension developers with domain expertise write extension specifications that define domain-specific language features with new concrete syntax and semantic analyses. The syntax and semantics are specified declaratively via self-contained extension specification. To aid modularity, extensions that add new constructs are expected to specify how equivalent host code is to be generated. Thus generated compilers are source-level translators from extended to host code. The generated host code can be compiled with traditional compilers to generate valid executables. Ideally, extensions would provide efficient translations to valid host language code, and feedback to the user in the form of high-level domain-specific error messages.

While modularity in extension specification is useful, composability is even more desirable. The library model allows for incorporating static analyses that flag potential problems, either at extension development time or composition time. Ideally, a statically validated extension specification could be freely composed with host specifications and with other validated extension specifications. An unsophisticated extension writer would be shielded from the more technical or difficult details of the framework, but would ideally be guaranteed that his validated extensions work as expected.

As extension complexity increases, the likelihood that static analyses will detect most run-time problems decreases. The extension writer might have to resolve concrete syntax conflicts, ensure analysis termination, or set priorities for optimizing transformations.
He would bear a larger share of the responsibility of ensuring safety and correctness.

**Specifying a Particular Extended Version of the Host Language** The user programmer would compose the host specification and extension specifications to automatically generate the extension 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 language by retrieving the appropriate extensions and composing them with the host specification. From the programmer’s point of view, extensions should be like plug-ins: easy to use, safe and with features that have the same look-and-feel as constructs in the host language.

### 1.2.1 An Example of the Library Model in Action

To clarify the roles, specifications and code that we have referred to in the library model of language extensibility, we look at an example of the model in action.

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

```java
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 1.1: Code written in an extended version of Java 1.4, and equivalent Java 1.4 code.
Programming in Java Extended with an Enhanced for Loop and Complex Types  We assume that a user programmer desires to solve a problem using a version of the Java 1.4 language extended with two features. The first feature adds complex numbers as a primitive type and overloads operations such as addition to allow the programmer to use them directly on com The second adds the java 1.5 enhanced for loop that provides nicer syntax for iteration over arrays and collections. Figure 1.1 shows an example of code written in a version of Java 1.4 extended with these features. This is followed by a figure showing equivalent code in Java 1.4, the code the programmer would write without the new features, and the code that is generated by the extended compiler mentioned above.

![Diagram](image)

Figure 1.2: An overview of the actors, specifications and code in the library model of extensibility.
Figure 1.2 shows the various components of a framework that implements the library model of language extensibility. It shows the different actors, specifications and code written and generated at various levels, in a situation where a pre-existing extensible specification of Java 1.4 is extended independently by two extension developers.

**The Java 1.4 Host Specification**  A host specification developer has written a specification `java.sv` of Java 1.4. Each self-contained specification is represented as a single file with extension `.sv` (although actual specification comprise multiple files). We look at the Java 1.4 host specification in detail in Section 2.3.

**The Complex Number and Enhanced for Extension Specifications**  Two separate extension developers have written extension specifications to the Java 1.4 host. The first developer’s specification `java_complex.sv`, adds complex numbers as a primitive type to Java 1.4. We look at this extension in detail in Section 2.2.2. The second, `java_foreach.sv`, adds a Java 5-style extended for loop. We look at this extension in detail in Section 2.2.1. Both these extensions have definitions that depend on the host specification (as indicated by the dotted arrows).

**Generating Compilers for Extended Versions of Java 1.4**  Two programmers with different programming needs specify different versions of Java 1.4 by specifying the host and extension specifications that are to be composed by the extensible language tool to generate the required translators. The users will use these generated compilers to translate the source code they write to pure and valid Java 1.4 code. The first user creates a compiler for Java 1.4 with the extended for loop by importing the `java_foreach.sv` grammar specifications into the `java.sv` host (indicated by solid arrows). We look at the process of composing extensions in Section 2.5. The tool generates the executable `java_foreach` which translates extended Java 1.4 programs `prog1.java_foreach` and `prog2.java_foreach` to the equivalent Java 1.4 programs `prog1.java` and `prog2.java`. The second user desires a version of Java 1.4 extended with both the enhanced-for and complex number extensions. He imports both extension specifications into the host specification. The generated executable `java_foreach_complex` translates programs `prog3.java_foreach_complex` and `prog4.java_foreach_complex`, written in the extended language, to the equivalent Java 1.4 programs `prog3.java` and `prog4.java`.
These are similar to the programs shown in Figure 1.1. The standard javac compiler is used to compile the generated Java programs into executable byte-code.

1.3 Challenges to Implementing a Modular Extensible Language Framework

There are challenges to implementing an extensible language framework that allows for the composition of modular extensions.

1.3.1 Issues with Specifying Concrete Syntax in an Extensible Fashion

The availability of more intuitive syntax is a major incentive to 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 and extension syntax. Parse tables generated by tools such as yacc are brittle in terms of whether conflicts emerge on the addition of new concrete productions.

There are also issues in the composition of lexical syntax specifications. A scanner cannot be generated if two extensions assign different terminals to the same token, or different precedences or associativities to the same binary operator. This is true even if syntactic context from the parser can be used to perform disambiguation, so that different tokens 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. Even more useful would be a modular analysis that would guarantee composability of independently developed concrete syntax specifications if they satisfy certain constraints.

In Section 1.4.2, we will briefly describe Copper, a solution to this problem developed by August Schwerdfeger [3, 4, 5]. The discussion of the concrete syntax aspects of Silver is included here to give the full picture of the Silver framework. We emphasize that this is not our work, but that of a member of the MELT group.
1.3.2 Issues with Specifying Semantics in an Extensible Fashion

There are challenges to developing a framework that allows for specifying semantics in an extensible manner. Extensions may specify domain-specific analyses for the constructs they add or new analyses for existing features. They may need to access and extend the host type system and environment. They may specify translations to equivalent source code, or the generation of optimized code based on static compile-time analyses such as data-flow analyses. The goal of extension composability means any framework for extensible semantics must ensure that extensions for new translations and new features work together.

There are many different approaches to language semantics. Each of these has its own set of challenges with respect to extensibility. Object-oriented frameworks, for instance, are well-suited to situations in which new constructs are added to an existing host. On the other hand, functional frameworks are suited to cases where new analyses are specified for existing constructs, but no new constructs are added. The expressive power of each approach must be balanced against the need for static safety analyses. Ideally the approach we employ would be well-suited to expressing programming language semantics via self-contained modular specifications that could be easily composed.

1.4 Silver: An Extensible Language Framework

Silver [6] is a general-purpose, high-level framework developed to implement the library model of extension development. The declarative Silver specification language can be used to specify the host concrete syntax and semantics, and independently, the extension syntax and semantics, often in terms of host semantics. Given a host specification and a set of extension specifications, the Silver tool constructs a generated compiler for an extended version of a host language.

The problem of brittle concrete syntax specifications is handled using the associated Copper tool that uses context-aware scanning to generate a parser for the extended language. The problem of specifying host and extension semantics (such as pretty-printing, error checking and source-level transformations) is handled via the evaluation by Silver’s attribute grammar evaluator-generator, of functions written in the higher-order attribute grammar formalism, on the nodes of the constructed syntax tree. Attribute grammar
fragments can be easily composed and provide a convenient means to specify semantics in a composable fashion.

Given a valid combination of specifications, the Silver extension composer composes the host and extension concrete syntax and semantics, resolves dependencies between them, 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 code.

1.4.1 Silver Specifications

![Diagram of Silver specifications]

Figure 1.3: Overview of the declarations in a Silver specifications.

The Silver declarations for a host or extension’s lexical and concrete syntax, and abstract semantics, are grouped into Silver files (with extension .sv) which are further
organized into grammar modules. Figure 1.3 shows an overview of the various declarations in a module. 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 its directory and defines a shared scope that includes all the declarations in the Silver files in that directory. It provides a means for other modules to import its declarations. Module names also incorporate Internet domains, in a manner similar to Java packages, to prevent name clashes. For example, the Java host grammar in Figure 1.1 would be specified in the Silver module `edu:umn:cs:melt:java`, and its declarations consist of the Silver files in the directory `edu/umn/cs/melt/java/`.

Modules group related declarations such as the attribute grammar fragments that define the semantics of a host language, or the terminals, non-terminals, context-free productions and precedences that define the concrete syntax of the host 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 a pre-existing host language module. Further details with examples of Silver specifications are provided in Chapter 2.

### 1.4.2 Extensible Concrete Syntax via Copper’s Context-Aware Scanning

Composing concrete syntax specifications which tend to be brittle in the face of modifications, is handled by the associated Copper tool, an integrated parser and context-aware scanner generator developed by August Schwedfeger [3, 4, 5]. The discussion of the concrete syntax aspects of Silver is included here to give the full picture of the Silver framework. We emphasize that this is not our work, but that of a member of the MELT group.

Copper reads in the Silver declarations for the 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 tools such as yacc. It uses 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 that verify that the concrete syntax definitions in an extension specification follow certain reasonable restrictions which guarantee successful composition (with respect to parser generation) with other well-behaved extensions.

1.4.3 Extensible Semantics via Higher-Order Attribute Grammars

The problem of specifying and composing host and extension semantics, is handled by Silver’s attribute grammar system. Attribute grammars [7] specify language semantics by associating values known as attributes to the nodes of the 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 instances are computed by retrieving and evaluating the appropriate attribute definitions from its production.

In higher-order attribute grammar frameworks 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 [8, 9]. This makes it easier to specify analyses such as type-checking and code generation. Host semantics is specified as a higher-order attribute grammar, which is evaluated using Silver’s attribute grammar system. Host 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 syntax-directed nature of attribute evaluation makes it a useful formalism for analysing code and tasks such as translation, optimization and error-checking. Using a well-defined, declarative and high-level formalism such as attribute grammars aids in the writing of modular and composable host and extension semantic specifications. Self-contained extension specifications can be written as separate grammar fragments, which can be statically analysed before being composed with the host and other extensions to generate the evaluator for the extended language. Finally, there are existing attribute grammar analyses and tools for constructing efficient and terminating evaluators.
1.4.4 Static Analysis of Silver Specifications for Composability

As we have seen, independently developed extensions may not be compatible with respect to syntax or semantics. For example, two extensions may independently compose validly with the host, 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, but when composed together, introduce circularities in the host type hierarchy. Thus it is essential for the framework to provide analyses on specifications to reduce problems during composition. The higher-level abstractions in Silver for specifying modular extensions must be accompanied by domain-specific analyses that increase the reliability of generated compilers by analysing extension specifications against host 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.

- Analyses would preferably detect problems modularly at specification development time, guaranteeing extension writers that their specifications will compose properly with other validated extension specifications. Problems detected at this stage can be solved by the extension writer, who presumably has the required domain knowledge and compiler expertise.
  
  For example, Silver’s attribute evaluator verifies that a given attribute grammar has valid occurs-on declarations, attribute definitions and aspect definitions. 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 [4]. These analyses and others such as the type checking performed by the Silver compiler provide modular guarantees.

- Less useful than modular analyses are monolithic analyses run at composition time on the extended compiler specification. These analyses would flag incompatible extensions, but user programmers would likely not have the needed access or expertise to resolve conflicts by modifying the host or extension specification. An example of a monolithic analysis is the higher-order attribution termination analysis presented in Chapter 4.
• Less useful than monolithic analyses are “run-time” analyses performed during the execution of the generated compiler, i.e., during the compilation of the extended language source code. For example, Silver’s lazy attribute grammar evaluator may in some cases detect undefined attributes only at run-time.

Non-Termination of Higher-Order Attribute Evaluation An issue of special significance for attribute-grammar frameworks such as Silver is that of improper termination of attribute evaluation during the execution of the generated compiler. A proper attribute evaluation sequence terminates after a finite number of steps with valid values assigned to every attribute instance. Attribution may terminate abnormally for a number of reasons, such as circularities in the attribute definitions, or missing definitions. There are well-known analyses that can statically check attribute grammars for circularities and definedness. Other problems include non-terminating calls to external functions. Finally, with higher-order attribute grammars, an infinite number of trees may be created during attribute evaluation even if the grammar is non-circular. This would cause attribute evaluation of undefined instances to continue indefinitely. As with any Turing-complete framework, automatically determining whether attribute grammar evaluation terminates, is undecidable. In Chapter 4 we will present a conservative analysis that statically checks higher-order attribute grammars for non-termination of tree creation during attribution.

The Need for Well-Written Host and Extension Silver Specifications The generated compiler cannot always be validated, as when two extensions independently specify global transformations whose results depend on the order in which they are applied. Thus some impediments to compositability may be beyond analyses of specifications. The host and extensions must therefore be designed and implemented keeping in mind the need for easy extensibility and safe composability, to increase the likelihood that generated compilers act as expected. In Chapter 2, we look at the process of designing the host type system and environment so that extensions can conveniently and safely access them, and add new types and typing relations. The use of Silver features such as collection attributes, aspects and forwarding is key for the extension developer to write specifications that are modular and composable. One can develop a set of best
practices that, if followed by extension writers, would increase the likelihood that their extensions work with other well-behaved extensions. It may be possible to develop static analyses that enforce these practices.

1.5 Contributions

The dissertation makes two contributions toward exploring the use of a declarative, attribute grammar-based tool (extended with features such as forwarding, aspects, higher-order attributes and collections) for writing modular, composable and statically analyzable language specifications. First, it gives a demonstration of the feasibility of the library model of extensibility by describing two examples of using Silver to write host and extension specifications for mainstream languages. Second, it presents an attribute-grammar analysis that statically checks for certain kinds of non-termination errors at compiler run-time, and thereby increases the reliability of the compilers generated by Silver from the host and extension specifications.

1.5.1 Extensible Silver Specifications for Real-World Languages

We describe ableJ, an extensible host specification for Java 1.4. ableJ can be used to generate front-end translators from code written in Java extended with various features, to code in pure Java 1.4. We describe the challenges involved in designing and writing the host specification so that its type system and environment can be conveniently accessed and modified by extensions. This allows for new types and type relationships to be safely added. Extensions written to ableJ range from a simple macro-like enhanced for extension, to more complex ones for complex number types, auto-boxing and algebraic data-types with pattern-matching.

We present a second extension example in which the Silver specification language is extended with constructs to specify and perform data-flow analyses. The extension allows for specifying 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 by attribute definitions to generate optimized source code.
1.5.2 A Static Analysis for HOAG Termination

As we have seen, attribute grammar evaluation may not terminate because of various reasons, such as circularities in attribute definitions, non-terminating function calls and infinite tree creation. We describe an analysis that statically checks HOAG specifications for infinite tree creation during attribution, and thereby increases the reliability of the compilers generated by the Silver tool. The analysis is monolithic since it is run at composition time. The analysis first attempts to construct a terminating set of rewrite rules to model attribution in the absence of inherited attributes. It then attempts to order the non-terminals based on how they decorate each other as inherited attributes. We formally prove that if both the rules and ordering can be successfully generated, then tree creation always terminates. The analysis is conservative since there are grammars that are terminating but cannot be shown to be so by the analysis.

The analysis is novel as it checks for non-termination due to the construction during attribution of an infinite number of attributed trees, and not circularities between dependencies. Combined with the circularity test, it provides the extension writer or user with a guarantee that the resulting compiler will not fail to terminate on account of improper attribute evaluation. We have run the analysis on the ableJ host and its extension specifications to demonstrate that the analysis is powerful enough to show termination of real-world grammars.

1.6 Dissertation Outline

The thesis contributions are described in Chapter 2, Chapter 3 and Chapter 4.

- Chapter 2 describes ableJ, an extensible host specification for Java 1.4. This work was published at the 21st European Conference on Object-Oriented Programming (ECOOP) in 2007 [10]. We were the main contributor to this work.

- Chapter 3 describes a second extension example in which the Silver specification language is extended with constructs to specify and perform data-flow analyses. This work was published at the 5th International Workshop on Compiler Optimization Meets Compiler Verification (COCV) in 2006 [11]. We were the main contributor to this work.
• Chapter 4 describes an analysis that statically checks HOAG specifications for infinite tree creation during attribution.

• Research on the design and development of the Silver tool, research on which was published at the 7th International Workshop on Language Descriptions, Tools and Applications (LDTA) [6] in 2007. While we played a significant role in this work, our contribution was less than that of the other authors.

• Chapter 5 looks at related work in the area of extensible languages and attribute grammar frameworks, and concludes.
Chapter 2

An Extensible Specification for Java

2.1 Introduction

In this chapter, we demonstrate the extensibility technique described in Chapter 1 on a mainstream language. We describe ableJ, a Silver specification of Java 1.4 [12], to show 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. This allows programmers to create customized extended languages that are adapted to the domains and problems in which they are programming. More importantly, the composition of language extensions is done by a programmer with no programming language design or implementation expertise. It is done automatically by the Silver tool.

In this chapter, we describe the host specification, focusing on design aspects that allow extension writers to safely and conveniently interact with the Java type system and environment. This allows for rich domain-specific semantic analyses that can generate efficient source code, and provide useful domain-appropriate feedback to the programmer. We describe a few illustrative extensions that extend the Java 1.4 host language. These extensions range from simple macro-style extensions that augment Java 1.4 with Java 1.5’s enhanced-for loop, to a more complex SQL extension that adds embeds the SQL query language into Java and checks queries for syntactic and type errors at compile time. We describe the various challenges to this ideal of writing modular composable extensions to a mainstream language, and describe some of our approaches to dealing
with them.

2.1.1 Bridging the Programming Semantic Gap

One problem that programmers must overcome when writing software for use in particular domains is the semantic distance between any high-level domain-specific abstractions that they have knowledge of, or expertise in, and the non-domain specific general features available for programming in most programming languages. Programming language do provide facilities for developing new abstractions, such as higher-order functions, classes, generics and parametric polymorphism. These abstractions may implement the desired functionality of the abstraction but do not provide new language constructs with new and more suitable syntax.

Domain-specific languages (DSL), on the other hand, 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 reduces the semantic gap between the general programming language and the specific needs of the program. Another advantage of DSLs is that unlike general purpose languages, they can perform domain-specific optimizations and other analysis. The drawback of the DSL approach is that it is often not economically viable to construct entire new languages and compilers, with associated tools if the programmer base is not large. And even if DSLs are available, it is often the case that a given programming problem spans more than one domain, compounding the viability issue.

It is unlikely that existing programming languages provide all the features (whether general purpose or domain specific) that might be need to solve a given programming problem. Programmers are thus forced to write their solutions in a form restricted by the idioms provided by the general purpose language. Using low-level representations is cumbersome, prone to errors and not easy to maintain.

2.1.2 Writing Composable Extensions in Silver

As described in Chapter 1, the Silver extensible language framework facilitates the writing of modular extension specifications that are also composable, i.e., extensions developed independently that can be composed together with the host to generate a working
compiler for an extended language. If extensions can be safely composed, then the user programmer, who has no knowledge of the extension implementations, can select the list of features he desires, and then use the Silver tool to automatically generate the compiler for the extended version of the language. For example, a programmer who wishes to solve a problem using complex numbers, in which he must retrieve complex number data from a relational database, could import both 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. Thus the Silver framework distinguishes between the process of language extension implementation performed by a feature designer with domain-expertise, and that of choosing a set of extensions to import into a host specification to automatically generate a compiler for the desired version of the host language. This is similar to the difference between the library writers, and programmers who import and use these libraries.

2.1.3 ableJ: A Silver Tool to Add Features to Java

In this chapter, we describe ableJ: an extensible language framework for Java 1.4 that aids in reducing this semantic gap while programming. Specified using the Silver tool, ableJ is an extensible language framework in which language semantics are specified using the higher-order attribute grammar formalism [8]. The extensible host language is specified as a higher-order attribute grammar. Extensions are similarly specified as fragments of higher-order attribute grammars. 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 such as compile-time source-code optimizations. This allows programmers to construct new customized domain-adapted versions of Java by importing the unique combination of extensions that will serve their programming needs, and thereby raise the level of abstraction in which their solutions are encoded. Extensions specify translations for any new constructs, to equivalent host language code in pure Java 1.4, which can be compiled by existing compilers to executable byte-code.

For example, Figure 2.1 shows code written in a version of Java 1.4 extended with a complex number type as well as a Java 5-style enhanced for loop. Also shown is equivalent code in Java 1.4 generated by a Silver constructed compiler, that can be
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 2.1: Code written in an extended version of Java 1.4, and equivalent Java 1.4 code.

compiled into byte-code with the Java compiler javac.

In Silver, attribute grammar fragments that define the host or extensions are packaged into Silver modules that are collections of Silver files. The Silver tool constructs the compiler for the extended language desired by the programmer by composing the host module and a list of extension modules, specified by the programmer. Host and extension attribute grammar specifications in the modules are combined to create an attribute grammar for the extended language. Silver includes an attribute grammar evaluator that evaluates this grammar to implement the compiler for the extended version of the language. Silver also allows for host and extension specifications to define concrete syntax in an extensible fashion. These concrete syntax specifications are similarly composed to generate the information required by Silver’s parser and scanner generator.

2.1.4 Reusing Host Semantics to Write Modular Extensions

Constructs that are defined as language extensions must have the same “look-and-feel” as constructs in the host language. Their syntax should be fit comfortably with the host
syntax. Further, the generated compiler should generate errors in terms of the higher-level constructs used by the programmer. They should not be reported in terms of the generated host language translation (as happens with macros). Finally, extensions should specify efficient translation for new constructs.

The concept of forwarding [13], an addition to higher-order attribute grammars, is an important means by which Silver allows for modular language definitions. It allows for extensions that meet the “look-and-feel” conditions above. Using forwarding, a newly-defined construct’s semantics can be specified implicitly by translation to equivalent host constructs. As an example, a “query” in a version of Java embedded with SQL (described in Section 2.2.4) would be translated to a call to a method in the JDBC library. But importantly, forwarding also allows for explicit semantic specification, by allowing for attribute definitions on productions that forward to equivalent host language constructs. This allows for additional domain-specific analysis such as, for example, static error-checking on queries specifiable using the SQL extension.

Semantic analysis in ableJ is done solely on source code. Host and extension specifications do not define or manipulate byte-code. If an extension adds a new construct, it is expected to also define its translation to equivalent code in pure Java 1.4. The generated compiler is therefore 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 analysis on it, and then generates valid Java 1.4 code. This is then compiled using a traditional Java compiler to executable byte-code. Extensions can perform analyses to ensure that errors, such as type errors or access violations that can be detected at compile-time, can be caught to ensure that the host translation of extended code is not erroneous. The ableJ extensible language framework aims to shield programmers from having to analyse any generated Java code. Errors should be meaningful and reported on programmer-written extended code.

2.1.5 Chapter Outline

In this chapter we describe the ableJ framework by examining the design of the host and the process of writing and composing modular extensions. We begin in Section 2.2 by describing a few representative examples of the kinds of extensions the framework is
designed to specify. Then in Section 2.3 we briefly describe the host language specification, primarily its concrete syntax and the higher-order attribute grammar that defines its semantics. In Section 2.4 we look at the basic process of modular extension specification by showing how extensions may add new concrete syntax and use the concept of forwarding [13] to specify translations to existing host constructs. In Section 2.5, we look at how a user can specify the extensions that he desires to import into the host using the Silver tool.

In the rest of the chapter, we look in more detail at the host features that allow extensions to perform powerful and safe analyses that interact closely with the host semantics. Section 2.6 looks at how the host environment is designed to allow extensions to access program symbol tables, and also add their own environment items in a way that avoids clashes with other extensions. Section 2.7 looks at the design of the host type system and demonstrates how extensions may add new types and sub-typing relations to the existing type hierarchy, specify mechanisms for run-time conversion, and perform operator overloading. Section 2.8 concludes with a discussion of ableJ.

2.2 Sample Language Extensions

In this section, we provide a representative list of extensions that have been written to the ableJ Java 1.4 host. The extensions include:

- an extension that extends Java 1.4 with constructs to specify and execute queries in the data-base query language SQL,
- an extension that adds Pizza-style algebraic data-types with pattern-matching, allowing for coding in a functional style,
- an extension that adds a new primitive type for complex numbers,
- an extension that adds Java 1.5’s enhanced for loop to Java 1.4, and
- an extension that performs automatic boxing and un-boxing between primitive and object numeric types.

This list is not exhaustive. Extensions not described here include:

- an extension to perform dimension analysis [14],
- an extension that adds concrete syntax for list values,
• an extension that adds notations for modeling languages [15, 16], such as condition tables from the modeling language RSML\textsuperscript{e}, that help in writing and reading complex boolean conditions, and

• an extension that extends Java 1.4 with constructs for programming in the domain of computational geometry, and uses overloading to statically generate optimized source code [17]. This extension augments ableJ with new numerical types that add unbound-precision integers to Java. Unlike general purpose languages and library-based implementations, the computational geometry extension uses domain-specific knowledge to optimize source code. When similar constructs were added to a small C-like language, the generated C language code was 3 to 20 times faster than similar code that used the CGAL geometric library, which is usually considered the best C++ template library for this area.

2.2.1 Adding an Enhanced for Loop

<table>
<thead>
<tr>
<th>Member[] group;</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>for (Member m:group) {</td>
</tr>
<tr>
<td>m.process ();</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Member[] group;</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>for (int i = 0; i &lt; group.length; i++) {</td>
</tr>
<tr>
<td>Member m = group[i];</td>
</tr>
<tr>
<td>m.process ();</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ArrayList group;</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>for (Member m:group) {</td>
</tr>
<tr>
<td>m.process ();</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ArrayList group;</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>for (Iterator it0 = group.iterator ();</td>
</tr>
<tr>
<td>it0.hasNext ();) {</td>
</tr>
<tr>
<td>Member m = (Member) it0.next ();</td>
</tr>
<tr>
<td>m.process ();</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Figure 2.2: Using the enhanced for construct to iterate over arrays and collections, and the equivalent Java 1.4 code that is generated.

The first extension we look at extends Java 1.4 with a construct identical to the enhanced for loop introduced in Java 1.5 [18]. This allows for iteration in source code
over arrays and over collections of discrete elements such as lists. This is an example of a simple and easy-to-understand extension that a programmer can add to Java without waiting for a major update of the compiler.

Figure 2.2 shows examples of code using both kinds of iteration. Also shown is the generated Java 1.4 versions of the code. While Java 1.5 provides the iteration facility on objects of type `Iterable`, equivalent Java 1.4 code requires the use of 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 array using a numerical index. The `for` extension interacts with the host type system and environment, to ensure that iteration takes place only over iterable types. It also uses the facilities Silver provides to construct higher-order abstract syntax trees and forward to them to generate valid executable host language code.

### 2.2.2 Adding Complex Numbers as a Primitive Type

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

```java
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 2.3: Code in a version of Java in which complex numbers are a primitive type, and the equivalent Java 1.4 code that is generated.

We have developed an extension that adds a complex number type as a primitive type to Java 1.4. Figure 2.3 shows code in a version of Java extended with this
type. The extension provides a new construct to declare complex-typed variables using the complex keyword. Further, complex literals can be specified using the syntax complex (real-part, imag-part). As shown in the figure, the extension translates these constructs to pure Java 1.4 that uses the Complex class.

Aside from providing for nicer syntax, the extension performs domain-specific type-checking to ensure, for example, that any complex literal is such that its real and imaginary components are valid double expression. And as shown in the figure, the extension also generates type-specific versions of operations such as addition. We will use this extension to illustrate how extensions can add new types and sub-type relationships to the ableJ host, and how they can perform operator overloading.

2.2.3 Adding Auto-Boxing and Auto-Unboxing

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

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

Figure 2.4: Code in a version of Java 1.4 with auto-boxing and auto-unboxing, and the equivalent Java 1.4 code that is generated.

We have written an extension that adds automatic boxing and un-boxing of primitive types to Java 1.4 [10]. Figure 2.4 shows Java 1.4 code that uses auto-boxing and un-boxing, and the equivalent Java 1.4 code that is generated using the compiler for the version of Java 1.4 extended with auto-boxing and un-boxing. This is an extension that interacts closely with the host type system to generate type-specific versions of statements such as assignment.
2.2.4 Embedding the SQL Query Language

```java
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 };
    }
}
```

```java
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 2.5: Code written in a version of Java embedded with the SQL query language, and the equivalent Java 1.4 code that is generated.

In this section we describe an extension that embeds the SQL query language into Java [19]. The extension adds new domain-specific syntax to Java. It also accesses and extends the symbol-tables defined in the ableJ host grammar to perform compile-time domain-specific type-checking.

Figure 2.5 gives a fragment of code in the 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 strings). This type information allows for compile time detection of syntactic errors in the 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, then this would
be detected as a compile-time type error.

The extension specifies equivalent pure Java 1.4 code that uses the JDBC library, as shown in Figure 2.5. Queries are constructed as Java Strings, 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.

2.2.5 Adding Algebraic Data-types and Pattern-Matching

```haskell
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 2.6: Haskell code defining an algebraic data-type for lists.

This section describes a general-purpose extension to Java that adds Pizza-style [20] constructs for algebraic data-types with pattern-matching. This allows one to program in Java using functional idioms, such as those in the Haskell program shown in Figure 2.6. 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.

The corresponding extended Java code (shown in Figure 2.7) 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
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 2.7: Functional-style code defining an algebraic data-type for lists.
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 described in Pizza [20]. For each algebraic data-type, the extended compiler generates an abstract class with sub-classes corresponding to each case, as shown in Figure 2.8. Sub-classes have fields and constructor parameters corresponding 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 head and a List field for the list 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.

2.3 The Host Language Specification

In this chapter, we look at the extensible host specification we have written for Java 1.4. We look at how the Silver language can be used to specify Java 1.4’s concrete syntax, and to write attribute grammar fragments that define its abstract syntax and any semantic analyses such as type-checking or identifier disambiguation. The specification is described in more detail in [10]. The full specification is available for download at melt.cs.umn.edu/software.html.

Silver Modules and Files  The ableJ declarations for Java 1.4’s lexical and concrete syntax, and abstract semantics, are grouped into Silver files (with extension .sv) which are further organized into grammar modules. 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 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 related
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 2.8: The equivalent Java 1.4 code that is generated for the extended code in Figure 2.7.
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 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 a pre-existing host language module.

We show snippets of files in this directory in Figures 2.9, 2.10 and 2.11. Each Silver file starts with a declaration of its grammar’s name, using the grammar keyword. This is followed by a list of import statements that allows 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.

2.3.1 Specifying the Java 1.4 Concrete Syntax

This section describes how Silver’s specification language can be used to define the terminals, non-terminals and productions of the Java 1.4 context-free grammar, from which the scanner and parser will be generated, either for the host or the host extended with new constructs. Silver handles concrete syntax specifications using Copper, an integrated parser and context-aware scanner generator developed by August Schwerdfeger [3, 4, 5]. We note that Copper is not our work, but that of a member of the MELT group.

The parser and scanner generator uses only certain Silver declarations, viz. those of concrete productions and 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 (such as abstract productions, attribute definitions, and non-terminals and terminals that appear only in abstract productions) are ignored during parser generation. One can, on the other hand, use concrete productions for semantic analyses by defining attributes directly on them.

We first describe how the host language lexicon is specified, and then how the rest of the concrete syntax is specified. Figure 2.9 shows a part of the specification of Java 1.4’s lexicon. The specification first uses the ignore terminal construct to define comment-type tokens that will be ignored by scanner and not sent to the parser. Non-comment
grammar edu:umn:cs:melt:ablej;

ignore terminal WhiteSpace /\t\n*/ ;
ignore terminal BlockComment
    /\s\*\*(\*\*\*+\*/\s\*\*)*/\*/ ;
terminal Lbrace_t ' ' ;
terminal Rbrace_t ' ' ;
terminal Lparen_t '(' precedence = 140, association = right;
terminal Rparen_t ')') precedence = 140, association = right;

terminal Semi_t ';';
terminal Eq_t '=' precedence = 10, association = right;
terminal Plus_t '+' precedence = 120, association = left;
terminal EqEq_t '==' precedence = 80, association = left;

lexer class keyword;

terminal Else_t 'else' lexer classes { keyword }, precedence = 160;
terminal If_t 'if' lexer classes { keyword };
terminal Int_t 'int' lexer classes { keyword };
terminal While_t 'while' lexer classes { keyword };

terminal Id_t /[a-zA-Z_\$][0-9a-zA-Z_\$/]* submits to { keyword };
terminal Intconst_t /([0-9]+|0[xX][0-9a-fA-F]+)[lL]*/ ;

Figure 2.9: Specifying Java 1.4’s lexicon in Silver.
Terminal symbols are defined using terminal keyword, in one of two ways. Symbols for fixed string keywords such as While_t and operator symbols are declared by specifying the string in single quotes. Such fixed string terminals can be represented directly by quoted strings in the grammar's productions, as shown in Figure 2.10. Variable terminal symbols, on the other hand, are declared using standard regular expressions. For example, the terminal symbol for identifiers, Id_t is declared to match all non-empty alpha-numeric sequences with an initial alphabetic character. Terminal symbol declarations may also specify operator precedence and associativity information that can be used by the parser to handle ambiguities in the grammar. The specification language also includes mechanisms to specify lexical precedence, so that keywords for example, have lexical precedence over identifiers. This is needed so that for example, the string "while", which is matched by the regular expressions of both identifiers and the While_t keyword, is matched only by the latter.

Figure 2.10 shows parts of the Silver specification that define its concrete syntax. The nonterminal keyword is used to specify the grammar's non-terminals. The non-terminals Expr, Type, Stmt and Stmts in the concrete grammar represent Java expressions, types, statements and statement sequences, respectively. Root represents a .java file. It is flagged as the grammar start non-terminal in the grammar's driver file using the parser construct, as shown in Figure 2.14.

The figure also shows some of the concrete productions in the Java grammar. The types of named parameters in production signatures are specified using the :: construct. Thus in the if_then_else production the third parameter is named cond and is of non-terminal type Expr. In Silver, concrete production declarations may also include attribute definitions, as described in the next section. These are elided in the figure. Other terminal, non-terminal and productions declarations define other Java constructs such as classes, methods and member declarations.

### 2.3.2 Specifying the Java 1.4 Semantics Using Higher-Order Attribute Grammars

ableJ's semantics is specified as a higher-order attribute grammar, which is evaluated using Silver's attribute grammar system. Host and extension semantics such as pretty-printing and type and binding checking are specified via attribute grammar fragments
grammar edu:umn:cs:melt:ablej;

start nonterminal Root;
nonterminal Expr, Type, Stmt, Stmts;

concrete production program
root::Root ::= '{' stmts::Stmts '}' { ... }

concrete production sequence
stmts::Stmts ::= first::Stmt rest::Stmts { ... }

concrete production stmt_one
stmts::Stmts ::= stmt::Stmt { ... }

concrete production if_then_else
if::Stmt ::= 'if' '(' cond::Expr ')' then::Stmts 'else' else::Stmts { ... }

concrete production while
loop::Stmt ::= 'while' '(' cond::Expr ')' body::Stmts { ... }

concrete production assignment
assign::Stmt ::= var::Id_t '=' expr::Expr ';' { ... }

concrete production declaration
dcl::Stmt ::= type::Type var::Id_t ';' { ... }

concrete production reference
expr::Expr ::= var::Id_t { ... }

concrete production int_constant
expr::Expr ::= int::IntConstant_t { ... }

concrete production equality
equal::Expr ::= lhs::Expr '==' rhs::Expr { ... }

concrete production plus
sum::Expr ::= expr1::Expr '+' expr2::Expr { ... }

concrete production int_type
type::Type ::= 'int' { ... }

Figure 2.10: Specifying Java 1.4's concrete syntax in Silver.
written using Silver’s attribute grammar specification language. The syntax-directed nature of attribute evaluation makes it a useful formalism for analysing code and performing tasks such as translation, optimization and error-checking. The use of a well-defined, declarative and high-level formalism such as attribute grammars aids in the writing of modular and composable host and extension semantic specifications. Self-contained extension specifications can be written as separate grammar fragments, which can be statically analysed before being composed with the host and other extensions to generate the evaluator for the extended language. Finally, there are existing attribute grammar analyses and tools for constructing efficient and terminating evaluators. In this section, we present a brief background on higher-order attribute grammars, before looking how Silver’s attribute grammar specification language can be used to define the host’s semantics. Later sections will look at how extensions can add their own semantic analyses.

Higher-Order Attribute Grammars

Attribute grammars [7] specify language semantics by associating values known as attributes to the nodes of the 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 instances are computed by retrieving and evaluating the appropriate attribute definitions from its production. Information flow between tree nodes results from writing definitions in terms of the values of occurrences on neighboring nodes.

In each step of the evaluation, an undefined attribute instance is selected and set to the evaluated value of its attribute defining expression. The selected instance must be such that any instances required for its evaluation must already have been defined. The process of attribution begins by evaluating attributes whose definitions are constants, or which depend on attributes such as `lexeme` which are set by the parser. The process evaluates the instances one at a time until there are no more evaluable attribute instances, either because all attribute instances have been evaluated, or because of circularities in the attribute definitions. If the grammar passes the circularity test, the latter case will not occur, and the process ends only when all attribute instances have been evaluated.
**Synthesized and Inherited Attributes** Attributes are of two types based on whether they primarily pass semantic information up, or down the syntax tree. *Synthesized attributes* implement semantic analyses which require information from sub-trees to be collected, passed up, and made available elsewhere in the program, such as definitions, code generation and error messages. 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. The synthesized instances on a node are defined on its production in terms of other synthesized instances on its children.

*Inherited attributes* are used when information needs to be passed down the tree. An example would be an environment passed down the tree for binding analysis. The inherited occurrences on a non-terminal may be defined by any production with that non-terminal on its right-hand side. Inherited instances on a node are therefore defined by its parent’s production.

**Higher-Order Attributes** In higher-order attribute grammar frameworks 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 [8, 9]. This makes it easier to specify analyses such as type-checking and code generation. Tree values are constructed using production names, production signature variables, terminal symbols, and accesses to other higher-order attribute instances. Productions can define local trees and use them during the evaluation of its other definitions to perform side-computations such as symbol look-ups.

During higher-order attribute evaluation, whenever a tree-valued instance is evaluated, a new tree with undefined attribute instances is created. Later evaluation steps will evaluate the attribute instances on this new tree. Evaluation ends when all instances on all trees are defined.

**Specifying ableJ’s Semantics in Silver’s Attribute Grammar Specification Language**

We now look at how we specify ableJ’s semantics using Silver’s attribute grammar specification language. Figure 2.11 shows some of the declarations that define abstract syntax
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 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 has not a duplicate declaration.
    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 2.11: Host specification declarations in ableJ.
and semantics for the ableJ language. We have elided some attribution definitions for clarity.

**Specifying Attribute Definitions in Silver Productions** Silver’s specification language includes constructs for declaring attributes and specifying attribute definitions within grammar productions. In the previous section we looked at how concrete productions are declared in Silver specifications. Attributes can be defined directly on these concrete productions. In the specification shown, the host semantics are defined on concrete syntax productions for simplicity. In the actual ableJ specification, we declare abstract versions of these concrete productions using the `abstract production` construct. Abstract productions such as `boolean_type_rep` are used only in the compiler’s internal analyses. They are not used in parser generation. An abstract production `skip` used only in generating valid translated code is also declared.

Each attribute declaration includes the kind (`synthesized` or `inherited`), the attribute type, and an optional list of non-terminals decorated by it. Occurrence relations are specified using either the `occurs on` or `with` construct. For example `pp` is a String-valued pretty-print attribute that evaluates to a string-version of its node’s sub-tree. `errors` is a list of error messages generated by its node’s sub-tree. Its evaluation is used to trigger most semantic analyses. Both these attributes decorate most non-terminals.

Silver’s attribute grammar specification language extends higher-order AGs with convenience features such as forwarding, aspect productions and collection attributes, which aid in the writing of extensible specifications.

**Forwarding** *Forwarding expressions* in productions are optional higher-order expressions (declared using the `forwards to` construct) that define a special tree value with the same root non-terminal [13]. During attribute evaluation, the value of any synthesized instance for which there is no explicit definition on the forwarding production, is instead queried off the forwarded tree. Forwarding is a means to add implicit definitions for some occurrences on a production (in terms of another production) while providing explicit definitions for all other occurrences. It therefore provides a way for extensions to specify translations to equivalent host code and re-use host semantics when needed, which aids in writing modular language specifications.
The use of forwarding is illustrated by how extensions define the Java 1.4 translations of their constructs. As shown in Figure 2.11, each non-terminal $X$ is decorated by a synthesized attribute $\text{host}\_X$ of type $X$ that stores the host translation corresponding to its sub-tree. Thus on the root production, $\text{host}\_\text{Root}$ 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 an equivalent sub-tree in the host. The definition of these attributes in conjunction with the semantics of forwarding, propagates the translation process at an extension construct down to child sub-trees. Thus querying $\text{host}\_\text{Root}$ off the root of a tree constructed in an extended version of Java 1.4, would return the host equivalent of the entire program.

**Aspects and Collection Attributes**  
*Aspect productions*, which are declared with the *aspect production* construct, are special productions that allow the extension writer to add attribute definitions to existing abstract or concrete productions. They provide a way to add new functionality (such as a new translation) to existing constructs by declaring new attributes and adding their definitions to existing productions.

*Collection attributes*, inspired by Boyland [21], are local attributes whose values are computed from individual values defined in multiple productions. They are accessible within any aspects to the production in which the local is defined. Their initial values are specified in their (non-aspect) productions, using a special $:=$ assignment operator. Extensions can contribute elements to collection attributes via aspects to these original productions using a special $\leftarrow$ assignment operator. The final evaluated value of a collection attribute is computed by gathering together all these individual values using a “collection” operator, which is declared in the attribute declaration using the *collect with* construct.

For example, as shown in Figure 2.11, *errors* is declared as a collection attribute. Its collection operator is list concatenation. The host adds messages to the *errors* attributes on its productions, based on the results of its type-checking and binding analysis. Extensions can add their own analysis results and error messages as shown in the extension descriptions below.
Pattern-Matching on Trees Using Production Names  Silver also allows for pattern-matching using production names on tree values [22]. Non-terminals assume the role of algebraic data-types. 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. We will see examples of pattern-matching in Silver in the description of the host type system in Section 2.7.1.

Other Convenience Features in Silver  Silver allows for declaring polymorphic non-terminals and attributes with type variables [22], and includes special syntax for type-safe polymorphic lists. Reference attributes [23] evaluate to pointers to the roots of fully attributed trees and can therefore be used to access the values of synthesized instances on previously attributed trees. Finally, functions declared using the function keyword are similar to productions except that the typed 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 2.16 and 2.18.

2.4 Adding New Constructs

In this section we look at two examples of extensions to illustrate the process by which extension writers write Silver specifications to extend the Java 1.4 with new constructs and new domain-specific analyses. We show how the concrete syntax can be added to add constructs for an enhanced for loop and to add complex types as a primitive type. Extensions can specify their own domain-specific analyses. They can use forwarding to specify equivalent host translations for their constructs, so that host semantics can be reused.

2.4.1 Adding the Extended for Loop

Figure 2.12 shows part of the specification of the extension described in Section 2.2.1 that extends Java 1.4 with an enhanced for loop, similar to that introduced by Java 1.5. The specification first defines the concrete syntax for the new construct by declaring a concrete production for_each. The new syntax refers only to existing Java 1.4 terminal
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 sub-type of Collection.

    local attribute isArray :: Boolean;
    isArray = ...; - Check if array type.

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

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

Figure 2.12: Adding an enhanced for loop to Java 1.4. See Figure 2.2 for examples of generated code.
symbols, and so the extension does not declare any new terminal symbols. The extension declares local attributes directly on this concrete production to perform type-analysis, error detection and to generate equivalent Java 1.4 code. The production defines some existing attributes such as pp and errors explicitly, so as to provide useful user feedback. All other attributes are accessed over the forwarded-to tree, the structure of which depends upon which the iteration is over an array, if it is over a collection, or if instead there is a type error.

Code for iteration over arrays is defined by the local forOverArray which increments an integer index in the array during each loop iteration. Generated code for iteration over collections is defined by the local attribute forWithIterators which calls the Collection interface’s methods. If the group 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.

To aid in the explanation, we have omitted some optional features of the Java 1.5 enhanced for loop, such as variable modifiers and statement labels. These pose no technical challenges, aside from the extra concrete syntax needed to parse the statement labels.

2.4.2 Adding the Complex Number Type

As mentioned in Section 2.2.2, the complex number extension adds constructs that declare variables to be of complex type, and that specify complex literal expressions. As shown in Figure 2.13, the complex extension declares a new keyword Complex_t matching the string "complex". This keyword and a concrete production complex_type together allow for the new complex type to be used in variable declarations. complex_type forwards to references to a Java class Complex, the definition of which is bundled with the language extension. The new syntax for complex literal expressions is defined by the complex_literal production. complex_literal verifies that the literal’s real and imaginary components are valid double expressions, and forwards to a call to a constructor for the Complex class.
import edu:umn:cs:melt:ablej;

terminal Complex_t 'complex';

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

congrete 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 2.13: Adding complex numbers as a primitive type to Java 1.4.
2.5 Composing the Host and Extensions

The components of a specific extended version of Java 1.4 (i.e., the Java 1.4 host and extension grammars) are specified by importing the appropriate modules into a single Silver file. Figure 2.14 shows a specification that specifies a version of the Java 1.4 host extended with two extensions: one that adds constructs for specifying and executing SQL queries, and another that adds algebraic data-types. The extended grammar is specified by a driver file in the module

```
exports edu:umn:cs:melt:ablej;
imports edu:umn:cs:melt:ablej only Root;
parser extparser :: Root {
  edu:umn:cs:melt:ablej;
}
function main
IOVal<Integer> ::= args::[ String ] ioIn::IO {
  return driver (args, extparser, ioIn);
}
```

Figure 2.14: Composing the Java 1.4 host with the SQL and algebraic data-type extensions.

Host 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 and appropriate extension modules 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 non-terminals and productions are used only to generate the extended language’s attribute evaluator. They are not incorporated in its parser. This
file defines a special `main` function that acts as a driver to call the appropriate parser on the input file.

The Silver compiler reads in the Silver files in the referenced modules. It verifies that 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 jar files for the Copper parser-scanner and the attribute grammar evaluator, as well as a jar linker.

### 2.6 Accessing and Adding to the Host Environment

Semantic analysis requires information flow between different parts of the program's syntax tree. Type and access-level information from the declarations of fields, identifiers, methods and classes must be available during type-checking of expressions, for example. In attribute grammar frameworks, this kind of information is passed between the nodes of the abstract syntax tree using both synthesized and inherited attributes working in tandem. ableJ provides a set of attributes, non-terminals, productions and functions that facilitate the encapsulation of semantic information at the point of declaration into scoped lists of environment items, passing them around the tree and ultimately the retrieval of information when needed. The environment is designed to allow extensions to add their own environment items to it in the appropriate scope. If they define new types, they may also define entirely new kinds of bindings.

#### 2.6.1 EnvItem: A Data Structure to Store Declaration Information

Figure 2.15 shows some of the declarations in the host related to the creation and access of symbol tables. The basic units of binding information are environment items of type `EnvItem` which are a data-structure that binds identifier names to type representations (among other things). 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` uses the `var_binding` production to create a simple environment item from the lexeme and declared type's `typerep`. The list of environment items in a node's sub-tree is gathered by the synthesized attribute `defs`, as shown in the `local_var_dcl` production.
- The basic unit of binding information and a production to create it.
nonterminal EnvItem;
abstract production var_binding
envItem::EnvItem ::= name::String rep::TypeRep { ... }

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

- Construction of environment item 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 2.15: Constructing and gathering up environment items at the point of declaration.

2.6.2 The Scoped Environment as a List of Lists

Environment items are passed down via the inherited attribute env (declared in Figure 2.16) which, at each point in the program, specifies the set of definitions that are live at that point. In Java, the environment consists of a list of layered scopes, each with its own set of environment items. Thus environment items are collected into scoped symbol tables at those nodes that define new scopes. When a new scope is not created, the environment is passed unchanged down the tree, as in the while loop. Separate scopes 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 methods and inner class definitions which define their own local scopes. The addLocalScope function creates a new local scope on top of a pre-existing environment. While the specification shown here implements the environment using lists, it could be implemented using search trees as is conventionally done for faster access.

This is used by the enhanced for loop to append a new local scope (containing the group member variable) to the existing environment before passing it down to the loop body, as shown in Figure 2.17.
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 tree.

Concrete production while
loop::Stmt ::= 'while' '(' cond::Expr ')' body::Stmt {
  cond.env = loop.env;
  body.env = loop.env;
}

Figure 2.16: Constructing and passing scoped environments down the tree.

The loop body's environment has an extra local scope with the collection.

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 2.17: Adding to the environment in the enhanced for loop.
2.6.3 Variable Look-Ups

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

- Using the look-up function 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 2.18: Looking up variables in the environment.

Finally, the ableJ specification includes a set of functions (such as lookUpVariable) to retrieve information from the environment based on identifier lexeme. Figure 2.18 shows how the identifier reference production uses the lookupVariable function to look up its identifier’s lexeme in its environment. An error message is generated if the identifier cannot be resolved to a single valid declaration.

Using Pattern Matching to Retrieve Environment Items to Prevent Errors

Figure 2.17 shows how extensions such as the enhanced for loop can easily add new binding items to the environment. The enhanced for extension adds environment items using existing constructs in the host. However, extensions can define their own productions to construct environment items. For example, the SQL extension adds definitions for table columns in its import table construct, which are then accessed in its using ... query construct.

Since a user might compose the ableJ host with both the enhanced for and SQL extensions, it is important that the environment be defined correctly in such a scenario. An SQL query inside an enhanced for loop must be able to properly extract the column
definitions added by a preceding import table construct. This is possible since the SQL extension defines a new production (say table_type_rep) to construct its environment items. The extension uses pattern-matching (on the production name) on any items retrieved from the environment to access any required column values. The new production table_type_rep is not known to any other extension. Thus any associated type representation information is not available to other extensions. While other extensions may (improperly) remove the SQL extension’s bindings, they may not retrieve or modify them, thereby causing subtle unexpected behavior within the SQL extension’s constructs. Thus any values that are added by a construct in one extension are properly accessible by other constructs in that extension. This is so even if the bindings are passed between these constructs via a construct in another extension. While it is true that an extension might improperly remove an element from the environment, the error will probably be dramatic and quickly detected.

2.7 Accessing and Extending the Host Type System

This section looks at issues 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 has been designed to provide hooks for extensions to interact with the host type system. It allows extensions to examine the resolved types of expressions and perform operations such as sub-type checks on types to ensure that expressions are correctly typed. Extensions can also add new types, such as a complex number type. They may introduce new concrete syntax to allow for variables to be declared in the new type, and to allow for the specification of its literals. Extensions can specify how their types interact with existing types, for example by defining sub-typing relations with standard Java types. They may specify whether existing operators are overloaded to handle operands in the new type. If values are to be automatically coerced between types in contexts such as assignments, code generation for automatic coercion can also be specified.
grammar edu:umn:cs:melt:ablej;

nonterminal TypeRep with name;

- Retrieve a type representation by name.
  function getTypeRep TypeRep ::= name::String { ... }

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

- Check 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 2.19: Type representations in ableJ.
2.7.1 Types and TypeReps

Java type-specific information is stored as trees rooted with the non-terminal symbol \texttt{TypeRep} with declarations as shown in Figure 2.19. The ableJ specification provides a few helper functions to perform type-checking. TypeRep trees are created from resolved type expressions (i.e., type expressions in which all identifiers have been associated with declarations), via abstract productions such as \texttt{double\_type\_rep} and \texttt{array\_type\_rep}. These productions take in type-specific information as children. Thus \texttt{array\_type\_rep} takes the TypeRep representing the array element type as a child. On the other hand, since \texttt{double\_type\_rep} has no extra information, one tree suffices for every reference to the double type.

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

abstract production complex\_type\_rep
typerp::TypeRep ::= { ... }

concrete production complex\_type
typte::Type ::= 'complex' {
    type.typeRep = complex\_type\_rep ();
}
```

Figure 2.20: Adding a new type for complex numbers.

Extensions can add new types. For example, the complex type extensions specifies that the complex type's internal representation is constructed using the abstract production \texttt{complex\_type\_rep}, as shown in Figure 2.20.

Non-terminals such as \texttt{Type} and \texttt{Expr} with associated type information are decorated with the attribute \texttt{typeRep} of type \texttt{TypeRep} as shown in the \texttt{double\_type} and \texttt{reference} productions in Figure 2.21. This attribute will be used during type-checking to retrieve and analyse expression types. Type-specific information can be stored on TypeRep nodes with extra attributes, such as a synthesized attribute \texttt{component\_type\_rep} on array types. During type-checking, this information could be retrieving using a set of “flag” attributes on TypeRep. However, we have implemented a pattern-matching extension
grammar edu:umn:cs:melt:ablej;

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

- Types and expressions are decorated with typeRep.

concrete production double_type
type::Type ::= 'double' {
   type.typeRep = double_type_rep ();
}

concrete production reference
eexpr::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 2.21: Type-checking in ableJ.
based on production name to the Silver specification language [22]. This allows this
information to be accessed in a much more convenient and functional-style manner using
the case construct. Thus the while production uses pattern-matching to verify that the
type of its condition expression is boolean.

```plaintext
import edu:umn:cs:melt:ablej;

congcrete production for_each
foreach::Stmt::= 'for' '('type::Type var::Id ':' group::Expr')' body::Stmt{
  - Using pre-defined functions 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 2.22: Type-checking in the enhanced for extension.

The enhanced for extension also uses pattern-matching on the collection’s type
representation to generate appropriate host code, as shown in Figure 2.22. The type of
the iteration is determined in the for_each production via the local flags isCollection
and isArray, as shown in Figure 2.12. 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, which is retrieved using the helper function getTypeRep.

2.7.2 Specifying Sub-Types and Mechanisms for Run-Time Conversion

Sub-type relationships are implemented using the non-terminals and attributes shown
in Figure 2.23. The non-terminal SuperTypeInfo is decorated by attributes that store
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
typerp::TypeRep ::= {
  typerp.superTypes := [ ];
}

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

Figure 2.23: Specifying sub-type relations.
the sub-type and 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 augment its host-initialized value with their own contributions via aspect productions.

```plaintext
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 2.24: Specifying new sub-type relations with the complex type.

Since the Java double type has no super-types, the host double_type_rep production initializes superTypes to the empty list [ ]. The complex type extension adds a sub-typing relation to this list, as shown in Figure 2.24. Since the complex type is a super-type of the existing Java 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 results in the automatic coercion of double-typed expressions 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 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 thus provides a mechanism by which the extensions can link new types such as complex numbers to the existing type hierarchy.

```plaintext
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 2.25: Using collection attributes to define specialized assignment trees.

The sub-typing information in the SuperTypeInfo non-terminals can be used to perform sub-type checks. This is done by calling the sub_type_check production (shown in Figure 2.25) 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 which decorate SuperTypeInfo and SubTypeResult. exprToConvert is set to the input expression while convertedExpr is the output type-converted expression.

```plaintext
forwards to if subTypeResult.isSubType
    then converted_assignment (lhs, subTypeResult.convertedExpr);
    else converted_assignment (lhs, rhs)

local attribute subTypeResult :: SubTypeResult;
subTypeResult = sub_type_check (lhs.typeRep, rhs.typeRep);
subTypeResult.exprToConvert = rhs;
```

Figure 2.26: Use of run-time conversion in the host assignment statement.
The fragment of code in Figure 2.26 shows how the host grammar uses this facility in the assignment production. 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 by 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 done using a local attribute subTypeResult. Local attribute instances are created for specific productions, rather than non-terminals. They evaluate to attributed trees whose roots can be given inherited attributes by the production on which the local is defined. In this example, the value of the inherited attribute exprToConvert 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 that defines the local. The full assignment production (described in Section 2.7.3) incorporates this sub-typing check into a more general provision for operator overloading.

The sub-type check using each type’s superTypes assumes that there are no circularities in the type hierarchy. While the host 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.

For example, assume that the host grammar has two unrelated types \( H_1 \) and \( H_2 \). Assume an extension \( E_1 \) that defines a new type \( X_1 \) where \( H_1 \) is a sub-type of \( X_1 \) which in turn is a sub-type of \( H_2 \). Assume a second extension \( E_2 \) that defines a new type \( X_2 \) where \( H_2 \) is a sub-type of \( X_2 \) which in turn is a sub-type of \( H_1 \). Each extension independently composes with the host without introducing circularities, but when combined, cause a sub-type relation to be circular.

### 2.7.3 Operator Overloading

Overloading operators allows programmers to write more concise and maintainable code. ableJ’s specification uses production collection attributes as a way to allow extensions to easily overload existing arithmetic and assignment productions. An example (shown in Figure 2.27) is the transformed attribute in the assignment production, which, in
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 2.27: Using collection attributes to define specialized assignment trees.

the absence of error, 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 off it. This provides a way to specify a specialized overloaded version of
the assignment statement. Both the host and extensions can contribute elements to
this collection attribute via aspects. For example, the auto-boxing extension generates
operations with boxed and un-boxed expressions in this way. As shown in Figure 2.28,
when the left-hand side is of type Integer and the right-hand side is a primitive inte-
ger, the assignment production forwards to a version in which the right-hand side is
a boxed integer. The ableJ specification includes similar productions for 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 spec-
ification language allows extensions such as the computational geometry extension to
specify optimizations for entirely new types. This distinguishes Silver from overloading
in C++.

At Silver run-time (i.e., during compilation of code in the extended language), every
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 2.28: Performing auto-boxing via collection attributes.

forwarded tree must be defined by a single value. Thus for type-specific dispatch to work, there must be exactly one element in the evaluated value of transformed. An extension adding a type must contribute one element to transformed if the two operands would otherwise not be related. By contrast, transformed may have multiple elements if two extensions independently specialize an operation such as assignment under overlapping pre-conditions. Silver’s collections do not perform compile-time checking to verify that attributes such as transformed have only one element. Such errors are only detected and flagged at run-time during compilation of extended source code, as shown in Figure 2.27. If there is an error, the production forwards to a skip statement.

2.8 Discussion

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 about the details of the implementation. We believe programmers must be able to compose language extensions in a similar way for them to have real-world impact. With this work we have demonstrated that one can build implement languages and extensions in a composable fashion. We have shown that programmers can build customized versions of their languages with the right domain abstractions, with no implementation knowledge of the host or extension,
or expertise in compilers and programming languages. Silver's syntax-directed attribute grammar formalism extended with features such as forwarding, aspects and collections is well-suited to writing extensions that are primarily syntax-directed. Forwarding provides a convenient way to specify code generation for both simple macro-type extensions that add new constructs such as an enhanced \texttt{for} loop, as well as those that add auto-boxing and algebraic data-types with pattern-matching and require more complex type analysis. We have demonstrated that it is possible to build a high-level host specification that contains provisions for extension writers to interact closely with the host environment and type system. We have had success in extending Java 1.4 with composable extensions that implement types for complex numbers, SQL queries and data, condition tables, computational geometry and dimensional analysis. These extensions define new types, sub-typing relations, mechanisms for run-time conversion and perform operator overloading. Thus extensions can be written in Silver in a way that allows for close interaction with the host while at the same time avoiding problems when composed with other extensions.
Chapter 3

Extending Silver for Data-Flow Analysis

3.1 Introduction

This chapter presents a second example of a useful non-trivial language extension. In this case the extension is written to the Silver specification language itself. It provides extension writers with constructs to specify and execute compile-time data-flow analyses (DFA).

A significant drawback in using AG-based frameworks like Silver for static analyses is the difficulty in specifying non-syntax directed semantic analyses such as optimizing data-flow analyses for imperative programs. These are more naturally specified on control-flow graphs (CFG). Temporal logics such as the Computational Tree Logic (CTL) provide a declarative framework with to specify and executing DFA conditions, by model-checking them on the CFG. In this chapter we describe an extension to the Silver specification language that

- provides constructs to specify how the CFG is to be constructed and labeled with attribute instance values, and to generate NuSMV version of the graph,

- extends the syntax of expressions to include CTL formulas, and

- allows attribute definitions to access the results of model-checking CTL formulas on specified CFG nodes using NuSMV.
Silver’s higher-order attributes can be used to construct the optimized version of the program, based on the results of executing the DFA.

3.1.1 Chapter Outline

This chapter is structured as follows. First, in Section 3.2, we look at the idea of specifying the data-flow side-conditions of program transformations as temporal logic formulas, that can be model-checked on the program control-flow graph. In Section 3.3, we discuss the idea behind constructing the nodes and labels of the program control flow graph from an abstract syntax tree in an attribute grammar framework. In Section 3.4, we describe a Silver extension for specifying and executing data-flow conditions during attribute evaluation. Section 3.5 concludes.

3.2 Specifying the Data-Flow Side-Conditions of Program Transformations as Temporal Logic Formulas That Can Be Model-Checked on the CFG

Dead-code elimination of unused assignment statements is a well-known example of a program transformation with data-flow side-conditions. The transformation performs data-flow analysis on the program to find assignments whose values are never used in every possible execution path. These assignments are removed in the optimized version of the program. Figure 3.1 shows the results of performing this transformation on a program in the small imperative language C-. C- is a subset of C that retains only basic statements, declarations and expressions. In the original program, the assignments to $y$ in line 3 and to $z$ in line 4, are “dead” as data-flow analysis shows that their assignments are unused in every 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 [24, 25]. In some cases, the data-flow side-conditions of program transformations can be specified as formulas in the Computational Tree Logic
{  
  int x, y, z;
  x = 0;
  y = 1;
  z = 3;
  if (x == 1)
    y = 5;
  else y = 2;
  while (y == 3)
    x = x + 1;
}

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

Figure 3.1: Performing dead-code elimination on a C– program.

Figure 3.2: The control flow graph of the program in Figure 3.1.
(CTL) [26], which can be model-checked on the program control-flow graph shown in Figure 3.2.

A CTL formula $f$ is checked on a state $n$ in a model $m$.

- $AX f$: is true on a state $n$ if $f$ is true on each of $n$’s successor states.
- $EX f$: is true on a state $n$ if $f$ is true on at least one of $n$’s successor states.
- $A [ f U g ]$: is true on a state $n$ 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 a state $n$ 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 a state $n$ if $f$ holds on all states reachable from $n$.
- $EG f$: is true on a state $n$ if $f$ holds on all states on at least one path from $n$.

Table 3.1: The temporal operators of the Computational Tree Logic.

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

$$assign: var := expr \Rightarrow \text{skip}$$

if $assign \models AX (A[! use(var) U (def(var) \land \neg use(var))] \lor AG \neg use(var))$

The transformation specifies that if a control flow graph node $assign$ (which is an assignment node) satisfies the CTL formula, then the node’s assignment statement is replaced with a $\text{skip}$ statement. The CTL formula is true on $assign$ if and only if the assignment to $var$ is dead, i.e., its value is never used in any future execution path. The meanings of CTL operators are provided in Table 3.1. Thus the formula states that on All $\text{next}$ states from the current assignment to $var$,

- either, on All paths, $var$’s lexeme is not used Until it is redefined without using the old value,
- or, on All paths, $var$’s lexeme is never (Globally) used.
3.3 Constructing the CFG Nodes and Labels from an AST in an AG Framework

In this section, we look at how the program CFG can be constructed on its AST. Figure 3.3 shows the abstract syntax tree of the program in Figure 3.1. Figure 3.4 shows how the control-flow graph’s nodes correspond to some of the nodes of the abstract syntax tree. Only a subset of syntax 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 that are relevant to the data-flow analysis that is to be performed on the model. Their instance values are therefore used to generate the labels for the nodes of the graph. Dead-code elimination requires assignment, skip and
Figure 3.4: Defining the CFG on the AST for the program in Figure 3.1.
conditional expression syntax nodes to have corresponding graph nodes. Declarations, types and similar non-flow related syntax nodes do not.

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, as shown in Figure 3.5, two attributes are declared and defined via aspects for these purposes. The String 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 3.4.1, we show how these attributes are used to generate node labels for the program CFG.

3.4 A Silver Extension for Specifying and Executing Data-Flow Conditions

There are obvious benefits to integrating these high-level representations into the Silver specification language so that declaratively specified data-flow properties can be model-checked on the program’s control flow graph for use in attribute evaluation, such as for the construction of optimization source. The extension described in this chapter adds constructs to Silver’s specification language that allow extension writers to specify compile-time transformations such as this. It adds constructs to allow the extension writer to specify, within his 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 used in their evaluation. In the generated compilers’ attribute grammar definition, 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 temporal logic formula as inputs. The results are retrieved and can be used in further attribute evaluation, such as for the construction of optimized code.
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 3.5: Specifying the labels on the nodes of the control-flow graph.
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 are shown using fragments of a Silver specification that adds compile-time dead code elimination to the C-language specification.

3.4.1 Specifying the Nodes, Edges and Labels of 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 3.6 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 AST nodes with corresponding control flow graph nodes are flagged using the \texttt{cfg nodes} construct. In this example, the non-terminals \texttt{Stmt} and \texttt{Expr} are flagged. Note that not all statements and expression nodes will have corresponding graph nodes. For example, declaration statements do not have corresponding CFG nodes. Specific graph nodes are created using the syntax \texttt{cfg syntax-node [successor-list]}. Each node is created from the corresponding syntax node and a list of successor nodes. Nodes have the type \texttt{CFG Node}, the declaration of which is automatically added by the extension.

The control-flow graph edges are defined by each node’s successor list. In this example, these lists are computed in aspects which define the synthesized attribute \texttt{entry} and the inherited attribute \texttt{successor}. For a syntax node, these attributes indicate the entry node and successor node, respectively, of the sub-graph corresponding to the syntax node’s sub-tree. In Figure 3.6, the \texttt{if.then.else} sub-tree has a corresponding CFG sub-graph in which the entry node is a new node constructed from the conditional expression’s syntax node. This entry node has the \texttt{entry} nodes of the \texttt{then} and \texttt{else} clauses as its immediate successors. Note that while a control-flow graph node may have multiple successor nodes, the CFG sub-graph that corresponds to an AST node’s sub-tree has only one successor node. The successor of the \texttt{if.then.else} sub-graph as a whole is passed to its children. It will be used during the creation of the other nodes
Figure 3.6: Specifying the construction of the control-flow graph for C- programs.
in its sub-graph.

Certain attributes are flagged as control-flow graph label attributes. Their instance values are used to generate the labels for the nodes of the graph. Control-flow graph nodes are labeled with the evaluated values of some of the attribute instances on their corresponding syntax tree nodes. Such attributes are flagged using the \texttt{cfg attributes} construct. In this example, both \texttt{def} and \texttt{uses} are flagged as label attributes. If a non-terminal is declared to be a node non-terminal, it must define the value of any label attribute that decorates it.

### 3.4.2 Model-Checking CTL Formulas with NuSMV to Generate Optimized Code

Our extension adds constructs for specifying data-flow properties as CTL formulas and checking them at compile time by calling the model-checker NuSMV \cite{27} 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 3.7, the NuSMV model for the graph is constructed using the syntax \texttt{smvmodel CFG-entry-node state-variable-range-list}. NuSMV models have the type \texttt{SMV Model}, the declaration of which is automatically added by the extension. NuSMV, like other finite state checkers, requires a list of ranges of possible values of all the state variables in the formula to be checked, in this case the label attributes \texttt{def} and \texttt{uses}. The \texttt{ranges over} and \texttt{ranges over powerset of} constructs are used to define the ranges of \texttt{def} and \texttt{uses} as the set of program variables, and that set's power-set, respectively. We have elided the definitions of the attribute \texttt{allVariables} which at the root node, contains all the variables in the program. The model is defined on the root and passed down in the inherited attribute \texttt{smvModel}.

The $\models$ construct is used to evaluate the results of model-checking a NuSMV CTL formula against a node in the model, with the syntax \texttt{NuSMV-model, CFG-node $\models$ SMV-formula}. The assignment production uses the $\models$ construct to model-check a CTL formula that states the condition under which the assignment is dead. The result of running the model-checking on the assignment node is used to perform a simple optional optimization that replaces the assignment with a \texttt{skip} statement if it is dead. The optimized version of the C- program is constructed using the \texttt{optimizedStmt} and
Figure 3.7: Constructing the NuSMV model to perform dead code elimination.
3.5 Discussion

The data-flow analysis extension to Silver described in this chapter 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 temporal logic that allows for writing data-flow conditions, which may 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 program analysis.
Chapter 4

Abstracting Higher-Order Attribution with Term Rewrite Rules

4.1 Simpler Abstractions to Model Higher-Order Attribution

Tree Creation May Not Terminate During Higher-Order Attribute Evaluation

The addition of higher-order attributes to canonical attribute grammars makes the problem of analysing evaluation more complex. Figure 4.1 gives the informal description of the process of higher-order attribute evaluation for a given parse tree $t$. In each evaluation step, an undefined evaluable occurrence (i.e., one whose required instances have all been evaluated) is selected and evaluated. If a local tree attribute is evaluated, its evaluated tree term is used to construct a new parse tree. Thus each newly created local tree adds new undefined attribute instances. Since there is no guarantee that only a finite number of trees will be created during evaluation, an infinite number of new undefined attribute instances may be added during evaluation, and attribution might not terminate.

Vogt [8] extended the notion of well definedness of attribute grammars to the class of higher-order attribute grammars by listing the conditions for attribution to terminate...
\begin{tabular}{|l|}
\hline
\textbf{SET} $T$ to $\{t\}$. \\
\hline
\textbf{WHILE} there is an evaluable attribute occurrence $n\#a$ in a tree in $T$ \\
\begin{itemize}
\item \textbf{IF} $a$ is a synthesized or local attribute \\
\hspace{1em} \textbf{SET} $e$ to $n\#a$'s defining expression $e$, specified in $n$'s production. \\
\hspace{1em} \textbf{ELSE} \\
\hspace{2em} \textbf{SET} $e$ to $n\#a$'s defining expression $e$, specified in $n$'s parent node’s production. \\
\item \textbf{SET} $v$ to the evaluated value of $e$. \\
\item \textbf{IF} $a$ is a local attribute \\
\hspace{1em} \textbf{ADD} an attributed version of the tree term $v$ to $T$. \\
\hspace{1em} \textbf{SET} $n\#a$ to the root of this tree. \\
\hspace{1em} \textbf{ELSE} \\
\hspace{2em} \textbf{SET} $n\#a$ to $v$. \\
\end{itemize} \\
\hline
\end{tabular}

Figure 4.1: An informal description of the process of attribute evaluation process for higher-order attribute grammars.
normally”, viz.

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

```verbatim
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 4.2: $G_0$: a complete, non-circular grammar for which tree creation does not terminate.

Vogt specifies versions of tests for completeness and circularity tests for higher-order attribute grammars. Attribution for a complete, non-circular attribute grammar will not terminate abnormally. But unlike canonical attribute grammars, attribution for a complete, non-circular higher-order attribute grammar may not terminate. For example, figure 4.2 shows a grammar that is complete and has no circularities. Attribute evaluation for this grammar does not terminate. The original syntax tree $p_R(p_X())$ evaluates the local $l_A$ to a tree $p_A()$, which evaluates its local $l_X$ to a tree $p_X()$, which creates another local tree $p_A()$, and so on indefinitely.

In this section, we look at developing an analysis to ensure that tree creation during attribution terminates. An analysis to guarantee that no evaluation sequences exist with
Using Rewrite Rules to Model Higher-Order Attribution

The problem of showing termination of higher-order evaluation during attribution is undecidable. Technically, it is equivalent to showing termination of the attribute evaluator program written in a programming language such as Haskell or Java. However, analysing lower-level implementations would discard a lot of higher-level domain-specific information about attribute grammars. We desire a simpler abstraction on which to perform analyses such as termination, one that incorporates our domain knowledge of attribute grammars as well as information gained from our implementations of attribute grammar specifications for real-world programming languages. Our experience with grammars such as ableJ shows us that sticking to a few simple and reasonable guidelines regarding higher-order attribute definitions, will guarantee termination of higher-order attribution.

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 not so imprecise that it cannot show termination for a large class of useful and interesting grammars.

We present an approach that uses existing termination analyses on term rewriting systems to show termination of higher-order attribute evaluation. Existing attribute grammar analyses such as the circularity test, and simplifying restrictions on grammar syntax and on the evaluation process, allow us to reduce the general problem of termination of attribution to showing termination of tree creation chains starting from the original tree, in which each non-initial tree is created as a local attribute on its predecessor.

We define a procedure to generate a set of rewrite rules for each grammar that models local tree creation during attribution, in the absence of inherited attributes. If the rules can be shown to be terminating (using existing TRS termination tools such as APROVE), then we have a guarantee of termination of tree creation, and thereby of evaluation in the absence of higher-order inherited attributes. The problem of non-terminating higher-order inherited attribute evaluation is handled separately, by imposing restrictions on the non-terminal occurs-on declarations that limit the number of
inherited accesses in any tree creation sequence.

1. IF the grammar does not satisfy the restrictions in Section 4.2, RETURN FAILURE.
2. IF the grammar fails the completeness test, RETURN FAILURE.
3. IF the grammar fails the circularity test, RETURN FAILURE.
4. IF the rules generated for the grammar as described in Section 4.4 cannot be shown to be terminating, RETURN FAILURE.
5. IF the non-terminals cannot be ordered as described in Section 4.5.2, RETURN FAILURE.
6. RETURN SUCCESS.

Figure 4.3: A decision procedure that statically analyses a grammar for termination of tree creation.

We could incorporate the rewrite rule generator and ordering constructor into a decision procedure that statically analyses a grammar for termination of tree creation, as shown in Figure 4.3. The procedure first checks if the grammar satisfies our restrictions on the syntax and attribution process; if it does not, the analysis cannot be used to check for higher-order termination. It then runs the higher-order completeness and circularity tests on the grammar; if the grammar is incomplete or circular, the termination analysis cannot be used on it. Then rewrite rules are generated for each grammar’s definition as described in Section 4.4, and checked for termination with an existing tool such as APROVE. If the tool determines that the rules are non-terminating, or if the tool cannot give an answer either way, the analysis fails. Finally, the analysis checks if the grammar’s non-terminals can be ordered to satisfy the conditions in Section 4.5.2. If the non-terminals cannot be assigned, or if no determination can be made about whether they can be ordered, the analysis fails. If the non-terminals can be ordered as desired, the analysis succeeds, and higher-order attribution always terminates.

This approach is feasible and provably correct, though conservative. As both APROVE and the ordering generator are conservative, the termination analysis as a whole is conservative. Thus there are grammars that are terminating, but for which the analysis will fail. The rules that we generate to model higher-order attribute evaluation rules
are simpler and easier to analyse than the corresponding Haskell function definitions. While they are approximate in that they may not be able to show termination for all terminating grammars, they are correct for our purposes. They can show termination for non-trivial grammars such as ableJ in a reasonable amount of time.

4.1.1 Chapter Outline

In this chapter, we show how the analysis would be implemented for a restricted class of higher-order attribute grammars similar to that defined by the Silver specification language.

- First, in Section 4.2, we define the syntax of attribute definitions and expressions in this class, and specify a formal model of attribution for it.
- In Section 4.3, we show how the problem of showing termination of higher-order attribution can be reduced to that of disproving infinite sequences of local trees.
- We then describe the process of generating rewrite rules for a grammar in Section 4.4.
- Finally, in Section 4.5, we order the grammar’s non-terminals to limit the number of inherited accesses in a tree creation sequence.
- We prove that if a non-circular grammar has a terminating set of rewrite rules and its non-terminals can be ordered as described, its evaluation always terminates. For clarity, some of the technical material about the analysis, including the proof of correctness, is given in Appendix A. This chapter gives a description of the analysis that focuses on the intuition behind it, and gives examples of its application.

4.2 A Restricted Class of Attribute Grammars and a Simple Attribute Evaluation Model

4.2.1 Restrictions on Attribute Definitions and Expressions

We start by describing the class of higher-order attribute grammars handled by our termination analysis. The design of this class is driven by two main goals.
• First, it is simple enough to conveniently illustrate the main ideas behind the analysis, but still includes most of Silver's important features. It is in fact expressive enough to specify grammars that are the equivalent of ableI.
• Secondly, as we will see in Section 4.3, the restrictions allow us to reduce the termination problem, to that of disproving infinite local tree creation sequences.

The formal definition of the attribute grammar class is given in Section A.2.1. As in standard attribute grammars [7], there are restrictions on which attribute occurrences may be defined on a given production in the grammar. Synthesized and local occurrences are defined on their nodes’ productions. Inherited occurrences are defined on the productions of their nodes’ parent, unless its node is the root of a tree evaluated as a local attribute. In the latter case, the inherited occurrences are defined on the productions of the local’s parent node, on which the local was evaluated.

Unlike standard grammars, definitions and expressions are further restricted in the following ways:

• For simplicity, we assume that there are no inherited occurrences on the root node symbol.
• A production may not refer to its own synthesized occurrences, or inherited occurrences on its children. Since it computes these values, there is no need to do so.
• Attributes may only be accessed off a node’s children, its locals, or its own tree (in the case of inherited attributes). All other kinds of accesses, such as nested accesses of attributes on attributes, are disallowed by the syntax. This allows us to isolate the creation of new syntax trees during evaluation to local attribute evaluation.
• For technical reasons, we do not allow a node to refer to its own tree (as a higher-order value) in any of its definitions. If such accesses are present, the rewrite rules generated by our analysis are always non-terminating. Our analysis therefore does not handle grammars with such accesses in their definitions. References to the tree to access attributes off it are allowed.
• All functions are primitive. They do not take trees as arguments or return them as results. They may therefore not be used in tree expressions, except in evaluating
the flags of conditional expressions.

- Forwarding is translated away using local attributes.

These restrictions do not limit expressiveness. Most Silver constructs that are not in this class can be rewritten to equivalent ones in the class using local attributes. This class is thus expressive enough to write grammars that are the equivalent of ableJ.

We now give two example grammars in the class of attribute grammars defined by these restrictions. The first grammar does not have inherited attributes; the second one does. We will use these grammars as running examples throughout this chapter.

**Example Grammar** $G_1$

Our first example grammar shown in Figure 4.4 implements a very simple imperative language. A program written in this grammar is shown below.

```plaintext
boolean x;
int y;
int z;
if (x)
  do y = z;
  while (x);
if (x)
  do z = y;
  while (x);
```

The grammar defines non-terminals and productions for statements, expressions and types. It computes a single pretty-print attribute on its productions. There are no inherited attributes. The concept of forwarding is illustrated in this grammar by how the ifThen production defines its 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. Tree creation in this simple grammar does terminate. Our analysis detects this as shown in Section 4.4.
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;
   }

-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));
  }

-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 4.4: $G_1$: a grammar for an imperative language without inherited attributes.
Figure 4.5: $G_2$: a grammar with inherited attributes decorating non-terminals that are 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 then 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 4.6: $G_2$: a grammar with inherited attributes decorating non-terminals that are
inherited attributes, part 2 of 3.
abstract production singleEnv e::Env ::= id::Id 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;
}

abstract production consTypeEnv te::TypeEnv ::= tid::Id 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::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 4.7: $G_2$: a grammar with inherited attributes decorating non-terminals that are inherited attributes, part 3 of 3.
Example Grammar $G_2$

Our second grammar example, shown in Figures 4.5, 4.6 and 4.7, also implements a simple imperative language, but uses inherited attributes to implement the program environment. It 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 with types. These type identifiers can then be used in variable declarations. A program in this grammar is shown below.

```c
typedef int t1;
typedef boolean t2;
int x1;
t2 x2;
x1 = x2;
```

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 a list of errors on each node. The grammar was written without a Type ::= Idₜ production to provide an example of inherited attributes decorating non-terminals that are themselves the types of inherited attributes. We look at the problem of tree creation in these kinds of grammars in more detail in Section 4.5. As shown below, our termination analysis detects that tree creation in this grammar does terminate.

4.2.2 The Evaluation Model

We now define the process of attribute evaluation for the restricted class of higher-order attribute grammars defined in the previous section. Attribute evaluation starts with a syntax tree, usually constructed by a parser, with all of its attribute instances undefined.

**Definition:** An attribute instance is *evaluable* in an evaluation state if all of the attribute instances needed to compute its value have already been evaluated.

In each step of the evaluation, an undefined evaluable attribute instance on one of the trees is selected and set to the evaluated value of its attribute defining expression. The
process of attribution begins by evaluating attributes whose definitions are constants, or which depend on attributes such as lexeme which are set by the parser. The process evaluates the instances one at a time until there are no more evaluable attribute instances, either because all attribute instances have been evaluated, or because of circularities in the attribute definitions. If the grammar passes the circularity test, the latter case will not occur, and the process ends only when all attribute instances have been evaluated.

We assume that during attribute evaluation, 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.

The semantics of local attribute evaluation is different from the semantics of other kinds of higher-order attribute evaluation (i.e., synthesized or inherited occurrences). When a local instance is evaluated, 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. Attribute evaluation terminates only when all instances on all trees are defined. Thus the process of local evaluation adds new trees during evaluation. Further, new trees are added only as local attributes, due to the restrictions in the previous section.

Our attribute evaluation model is simpler than most standard evaluators, including Silver's. This means that for a given syntax tree, while any attribute evaluation sequence possible in Silver is also included in this model, there are sequences in this model that are not possible in Silver. It is therefore possible that for an input tree, evaluation is non-terminating in this model, while it is terminating in Silver. This is another aspect of the conservative nature of the analysis.

We define notions of proper and improper evaluation as follows:

**Definition:** A *proper evaluation sequence* is an evaluation sequence that terminates with a valid attribution to every attribute instance in the original tree, and any tree created during evaluation.

**Definition:** An *improper evaluation sequence* is an evaluation sequence that is not proper, i.e., a sequence that terminates abnormally due to absent definitions, circularities in attribute definitions, non-terminating function calls, or other errors in the
For a non-circular, complete grammar with terminating functions, we can reduce the problem of improper attribute evaluation to non-termination of tree creation.

**Representing Evaluation State as a Tree of Locals**

Defining the evaluation model in this way lets us organize the set of syntax trees in each evaluation step into a *tree of locals* (TOL). This allows us to reduce the problem of non-terminating evaluation to that of non-terminating tree creation sequences, as shown in Section 4.3. A TOL is a tree in which each node represents a syntax tree. The root of the TOL is the original program syntax tree. A node’s child nodes represent all the local trees that have been created on any of the nodes of the syntax tree it represents.

We can generate trees of locals corresponding to the steps of an evaluation sequence. Every evaluation step can be represented by a TOL with nodes representing all its syntax trees (the original and all locals created), along with an attribution to all the instances in these trees. In the initial evaluation state, the TOL has one node, labeled with program syntax tree. The TOL of each later step is either the same as that of its previous step, or has one extra node and edge. 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. In all other steps, the TOL stays the same, though the attribution to an instance of an existing syntax tree changes.

An evaluation state’s syntax trees can be organized into a TOL since every local is created on an existing tree and every local is evaluated only once. Further, each TOL node has a finite number of children, as each syntax tree has a finite number of AST nodes each with a finite number of locals. This fact is used below to show that for an infinitely increasing sequence tree of locals, we can construct an infinite TOL path. We now look at two examples of how trees of locals are constructed during attribute evaluation.

**A Tree of Locals Example for Grammar \(G_1\)**

We show how a tree of locals is constructed during local attribute evaluation for the program shown on page 81 for
grammar $G_1$ shown in Figure 4.4. Since our model makes no assumptions about the order in which evaluable attribute instances are evaluated, we assume that local attribute evaluation takes place in the following sequence. At each step, we show the structure of the tree that is created. At the end we show the structure of the tree of locals at the point of evaluation of each of these steps. The root of the tree of locals is the program syntax tree.

\[ T_0: \text{The program syntax tree:} \]

```
1: consStmt
  2: varDcl
    3: boolType id(x)
  4: consStmt
  5: varDcl
    6: intType id(y)
  7: consStmt
  8: varDcl
    9: intType id(z)
  10: consStmt
```

```
11: ifThen
  12: varRef id(x)
  13: doWhile
    14: assign id(x)
    15: varRef id(y)
  16: varRef id(x)
  17: ifThen
  18: varRef id(x)
  19: doWhile
    20: assign id(x)
    21: varRef id(z)
  22: varRef id(y)
```
$T_1$: The first doWhile constructs its equivalent while tree (defined by the local fs on node 13 of $T_0$):

```
1: constStmt
  2: assign
     id(y)
     3: varRef
        id(z)
     4: while
        id(x)
        5: varRef
           id(y)
        6: assign
           id(z)
        7: varRef
```

$T_2$: The second doWhile constructs its equivalent while tree (defined by the local fs on node 19 of $T_0$):

```
1: constStmt
  2: assign
     id(y)
     3: varRef
        id(z)
     4: while
        id(x)
        5: varRef
           id(y)
        6: assign
           id(z)
        7: varRef
```

$T_3$: The first ifThen constructs its equivalent ifThenElse tree (defined by the local fs on node 11 of $T_0$):

```
1: ifThenElse
  2: varRef
     id(x)
  3: doWhile
     id(x)
  4: assign
     id(y)
     5: varRef
        id(z)
  6: varRef
     id(x)
  7: emptyStmt
     id(y)
```
$T_4$: The `doWhile` in the newly created `ifThenElse` tree constructs its own equivalent `while` tree (defined by the local `fs` on node 3 of $T_3$):

```
T4: 1: consStmt
    |   2: assign
    |     3: varRef
    |        | id(y)
    |        |     4: while
    |        |          5: varRef
    |          |      | id(z)
    |          |      | id(x)
    |          |      | id(y)
    |      6: assign
          |      7: varRef
          |        | id(z)
```

$T_5$: The second `ifThen` in the original program tree constructs its equivalent `ifThenElse` tree (defined by the local `fs` on node 17 of $T_0$):

```
T5: 1: ifThenElse
    |   2: varRef
    |     3: doWhile
    |          4: assign
    |            | id(x)
    |            |     5: varRef
    |              |      | id(z)
    |              |      | id(x)
    |        6: varRef
          |      | id(y)
```

$T_6$: The `doWhile` in the second `ifThenElse` tree constructs its equivalent `while` tree (defined by the local `fs` on node 3 of $T_5$):

```
T6: 1: consStmt
    |   2: assign
    |     3: varRef
    |        | id(y)
    |        |     4: while
    |        |          5: varRef
    |          |      | id(y)
    |          |      | id(x)
    |      6: assign
          |      7: varRef
          |        | id(z)
```

Assuming that local attributes are evaluated in this sequence, the tree of locals would be constructed as follows. Each node specifies the tree and the node of its parent on which it was created.
A Tree of Locals Example for Grammar $G_2$. In our second example, we show how a tree of locals is constructed during local attribute evaluation for the program shown on page 86 for the grammar $G_2$. Once again, the root of the tree of locals is the program syntax tree.

$T_0$: The program syntax tree:
$T_1$: The variable reference in the assignment statement constructs its environment (defined by the local ee on node 14 of $T_0$):
$T_2$: The `consEnvId` production in the newly created environment constructs its type environment (defined by the local `fte` on node 2 of $T_1$):

```
1: appendTypeEnv

2: emptyTypeEnv  3: appendTypeEnv

4: emptyTypeEnv  5: appendTypeEnv

6: constTypeEnv  9: appendTypeEnv

id(t2) 7: boolTypeRep  8: emptyTypeEnv

0: constTypeEnv 13: emptyTypeEnv

id(t1) 11: intTypeRep 12: emptyTypeEnv
```

$T_3$: The second `constTypeEnv` in the newly created type environment constructs its equivalent `appendTypeEnv` tree (defined by the local `fte` on node 10 of $T_2$):

```
1: appendTypeEnv

2: singleTypeEnv  4: emptyTypeEnv

id(t1) 3: intTypeRep
```

$T_4$: The first `constTypeEnv` in the type environment constructs its equivalent `appendTypeEnv` tree (defined by the local `fte` on node 6 of $T_2$):

```
1: appendTypeEnv

2: singleTypeEnv  4: emptyTypeEnv

id(t2) 3: boolTypeRep
```
$T_5$: The `consEnvId` production in the environment after resolving its type id constructs its equivalent `consEnvType` tree (defined by the local `fe` on node 2 of $T_1$):

```
   1: consEnvType
      2: boolTypeSep
      3: emptyEnv
```

$T_6$: The newly created `consEnvType` tree constructs its own equivalent `appendEnv` tree (defined by the local `fe` on node 1 of $T_5$):

```
   1: appendEnv
      2: singleEnv
      4: emptyEnv
      id(x2) 3: boolTypeSep
```

$T_7$: The other `consEnvType` in the environment constructs its `appendEnv` tree (defined by the local `fe` on node 5 of $T_1$):

```
   1: appendEnv
      2: singleEnv
      4: emptyEnv
      id(x1) 3: intTypeSep
```

At the end of this sequence of local evaluation, the tree of locals is as follows. Each node specifies the tree and the node of its parent on which it was created.

```
   T_0
     T_1 on 14
       T_2 on 2
       T_5 on 2
       T_7 on 5
         T_3 on 10
         T_4 on 16
         T_6 on 1
```
4.3 Reducing the HOAG Termination Problem to Disproving the Existence of Infinite Tree Creation Sequences

We can reduce the problem of guaranteeing termination of attribution to that of disproving the existence of infinite tree creation sequences by showing the following:

- Any improper evaluation sequence for a non-circular grammar is infinite.
- Any infinite evaluation sequence contains an infinite number of local tree creation steps. For such a sequence, we can construct trees of locals with increasing numbers of nodes and edges.
- We can then construct an infinite path of locals from the original tree, in which each non-initial tree is created as a local on its predecessor.

The restrictions on the grammar syntax and our assumptions on the evaluation model imply that any improper evaluation sequence for a non-circular grammar contains an infinite number of steps. This is because at every step of a valid evaluation sequence, either every tree is attributed, or there is an evaluable attribute occurrence, because the definitions are non-circular. This means that evaluation can continue at every step. An improper evaluation sequence therefore does not end and has an infinite number of steps.

For non-circular grammars in which an order exists to evaluate all attribute instances, evaluation terminates validly only if all instances on all trees are evaluated.

Since every evaluation step evaluates an undefined attribute instance, any infinite sequence evaluates an infinite number of attribute instances. As each tree has only a finite number of attribute instances, a sequence that evaluates an infinite number of attribute instances must create an infinite number of trees. New trees with undefined attribute instances are only created in local creation steps. This implies that an improper (and therefore infinite) evaluation sequence contains an infinite number of local tree creation steps.

For an evaluation sequence with an infinite number of local tree creation steps, we can construct an infinite list trees of locals, with increasing numbers of nodes and edges. Each of these tree of locals is rooted at a node representing the original syntax tree, and each of whose nodes has a finite number of children. König’s Lemma states that an infinite tree in which each node has finite children, has an infinite path. We can use
the lemma to construct an infinite path of locals - an infinite tree creation sequence - starting from the original tree, in which each non-initial tree is created as a local on its predecessor.

**Definition:** A *tree creation sequence* at an evaluation step is any sequence of trees starting from the program syntax tree, in which each non-initial tree is the value of a local that has been evaluated on its predecessor. A tree creation sequence at an evaluation step corresponds to a path in the tree of locals at that step.

Thus for any infinite evaluation sequence, we can construct an infinite tree creation sequence. And as we shall see, on any path in the tree of locals, there is a corresponding rewrite sequence.

**Theorem I:** For a tree in a non-circular grammar with terminating functions, if there is an improper evaluation sequence, there is an infinite tree creation sequence from the original tree, in which each successive tree is created as a local value on its predecessor.

A formal proof of **Theorem I** is given in Section A.3.1.

### 4.4 Rewrite Rule Sequences as a Simple Abstraction of Tree Creation Sequences in the Absence of Inherited Attributes

We have shown how for a non-circular grammar in our restricted class with terminating functions, we can reduce improper evaluation to infinite tree creation. In this section we use rewrite rules to model tree creation so that termination of the rewrite rules implies termination of all tree creation sequences.

The rules are generated for each grammar from the definitions of higher-order attributes in the grammar’s productions. The rules can be used to derive each evaluated local from its parent’s sub-tree. This can be used to generate a rewrite sequence that models an entire tree creation sequence, that is at least as long as the tree creation
sequence. Thus if the rewrite rules terminate, then no infinite tree creation sequence exists, and tree creation and therefore attribute evaluation always terminates.

**Theorem II:** If the rewrite rules are terminating, there is no infinite tree creation sequence, and hence no improper evaluation sequence.

We use existing tools such as APROVE which check term rewriting systems for termination to see if the generated rules terminate.

Before looking at the generated rules in detail, we give an example of a tree creation sequence and how the rules generated for its grammar can model it. Consider the final tree of locals on page 92 for the program on page 81 in grammar $G_1$ shown in Figure 4.4. An example of a tree creation sequence, defined by a path in the tree of locals, is $T_0, T_3, T_4$. The trees in this tree creation sequence are the original program syntax tree, followed by an `ifThenElse` tree, and finally, the tree forwarded to by the `doWhile` tree.
The rules generated for grammar $G_1$ defined in Figure 4.4 are as follows:

- $\text{doWhile} \left( s_1, e \right) \rightarrow \text{consStmt} \left( s_1, \text{while} \left( e, s_1 \right) \right)$
- $\text{ifThen} \left( e, s_1 \right) \rightarrow \text{ifThenElse} \left( e, s_1, \text{emptyStmt} \left( \right) \right)$

The rules derive each local tree from its parent’s sub-tree as follows:

Since each local’s parent is a sub-term of its predecessor in the tree creation sequence, we can generate a rewrite sequence for the entire tree creation sequence as follows:
boolean x;
int y;
int z;

ifThen
  id(x)
doWhile
  assign
    varRef
      id(x)
y = z;

ifThen
  id(x)
doWhile
  assign
    varRef
      id(x)
y = z;

ifThenElse
  varRef
    id(x)
  doWhile
    assign
      varRef
        id(x)
y = z;

emptyStmt

...
We now look at how the rules are derived from the production, how they model local tree creation from parent sub-tree to new tree, and how rewrite sequences can be constructed for entire tree sequences.

4.4.1 Generating Rewrite Rules from Higher-Order Attribute Definitions

Each rule is associated with a specific production. It rewrites tree terms with the production at its root (the local’s parent sub-tree) to evaluated local tree terms. All the rules for a production have the same LHS. The LHS term constructor and the number of rule variables are derived from its production signature.

Each rule has a different RHS based on the RHS of the higher-order attribute definitions:
• Explicit tree creation sub-expression 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.

• Accesses to synthesized occurrences on a child are represented by the child’s signature variable.

• Accesses to synthesized occurrences on local attributes are replaced by rewrite sub-terms generated from the local’s definition. Local attributes are thereby in-lined when generating rewrite rules; this is possible for non-circular attribute grammars.

• 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. The formal process of rule generation is described in Section A.4.1.

4.4.2 Constructing a Rewrite Sequence Corresponding to a Tree Creation Sequence

For each local tree creation step during evaluation, we can derive the evaluated tree value via a non-empty rewrite rule sequence from the local’s parent’s sub-tree.

\[
\textbf{Lemma} \ \omega: \text{Given a tree creation sequence (corresponding to a path in the tree of locals), every non-initial local tree can be derived via a non-empty rewrite rule sequence, from its parent’s sub-tree.}
\]

The intuition behind this can be understood by breaking the sequence into two parts, the first term, and the rest of the sequence. A formal proof of \textbf{Lemma} \ \omega is given in Section A.4.2.

The first term in the rewrite sequence represents the tree value after all conditions in its defining expression have been evaluated. The rules model conditional expressions
by generating multiple rules corresponding to the two clauses in the expression. The first term thus is a partially evaluated version of the expression in which all conditions have been evaluated, but in which attribute instance accesses are not computed and are represented by the sub-trees on which they are evaluated. This term is derivable from the tree term of the attribute instance's defining node, via the one rule that corresponds to the actual values of the conditional expressions.

In the first term in the sequence, 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 of the child or local tree terms, additional rewrites can be performed on these sub-terms to generate the actual evaluated attribute instance value. The fact that these additional rewrites can be performed to generate the correct value can be shown inductively on the structure of the defining expression. The inductive assumption is that all previously evaluated tree values have corresponding valid rewrite sequences from their parent sub-trees to the created values.

Given that we can construct sequences that derive each tree from its predecessor, we can construct a rewrite sequence that models an entire tree creation sequence.

\textbf{Lemma }\rho: \textit{Given a tree creation sequence (corresponding to a path in the tree of locals), we can construct a rewrite sequence of at least the same length.}

In a 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 (given by \textbf{Lemma }\omega) is a sub-term of the predecessor tree. Thus the predecessor tree itself can be rewritten to a term in which the parent sub-tree is replaced by the local. This new term contains the local as a sub-term, and therefore the process can be continued for the next evaluated local. A formal proof of \textbf{Lemma }\rho is given in Section A.4.3.

The rewrite sequence corresponding to each local’s evaluation is non-empty. We can therefore construct a rewrite sequence to model the entire tree creation sequence that is at least as long as the tree creation sequence. This tells us that if the rules terminate, then tree creation, and therefore attribute evaluation cannot continue indefinitely.
4.4.3 The Rules are Useful and Correct, Albeit Conservative

The problem of showing termination, whether for attribute evaluation or term rewriting systems, is undecidable, so our goal is an approximate solution. The design of the rules is guided by several factors. Since a constantly failing analysis would technically be sufficient in a conservative analysis, the generated rules must be useful and be to show termination of grammars such as ableJ in a reasonable amount of time. Finally, we must be able to formally show that the rules are correct for our purposes; if the generated rules are terminating, then there are no infinite tree creation sequence and therefore attribution in general terminates.

The rules we define here satisfy both these conditions. They are conservative in how they model tree creation since they derive more terms than are actually generated during higher-order evaluation. For example, attribute instance names are not retained in the sub-terms corresponding to synthesized attribute accesses. Thus the rules derive sequences corresponding to the evaluation of every possible higher-order synthesized instance on the child or in-lined local. Only one of these will corresponding 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. The generation of extra rules precludes the use of this approach to show termination of grammars with inductive function calls.

Figures 4.8 and 4.9 show an example of a grammar for which tree creation does not terminate. It is an extended version of Grammar $G_1$ defined in Figure 4.4 and defines the same language. In this grammar, a new attribute $\textbf{hostStmt}$ computes the “base” version of the program syntax, i.e., an equivalent version in which “extended” constructs such as $\text{doWhile}$ and $\text{ifThen}$ have been translated away. In practical usage, such an attribute would only be computed once, off the initial program tree. In the conservative evaluation model we have adopted, however, the $\textbf{hostStmt}$ 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 are non-terminating and are as follows:
Figure 4.8: $G_3$: a grammar for which tree creation does not terminate, part 1 of 2.
For "extended" constructs, hostStmt is computed off the forwarded to tree.

```java
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;
}
```

```java
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;
}
```

```java
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 4.9: $G_3$: a grammar for which tree creation does not terminate, part 2 of 2.

- varDcl $(t, id)$ $\rightarrow$ varDcl $(t, id)$
- assign $(id, e)$ $\rightarrow$ assign $(id, e)$
- while $(e, s_1)$ $\rightarrow$ while $(e, s_1)$
- ifThenElse $(e, s_1, s_2)$ $\rightarrow$ ifThenElse $(e, s_1, s_2)$
- emptyStmt () $\rightarrow$ emptyStmt ()
- consStmt $(s_1, s_2)$ $\rightarrow$ consStmt $(s_1, s_2)$
- doWhile $(s_1, e)$ $\rightarrow$ consStmt $(s_1, while (e, s_1))$
- ifThen $(e, s_1)$ $\rightarrow$ ifThenElse $(e, s_1, emptyStmt ())$

4.5 Ordering Trees to Limit Inherited Accesses in Tree Sequences

4.5.1 The Rules Do Not Work for Inherited Attributes

The term rewriting rules described above cannot model tree creation in the presence of higher-order inherited attributes. Rules containing just production names and child variables cannot accurately model tree creation with inherited attributes. The rules
cannot incorporate the contextual information required to evaluate inherited attributes
(attributes on the parent node).

We now give an example of a tree creation sequence with inherited attributes. The
rules for the grammar $G_2$ shown on page 86 are listed below. We have extended the
rewrite rules with a place-holder term INH to represent inherited accesses in the rewrite
term.

```
  • root (s) −→ s
  • root (s) −→ emptyEnv
  • root (s) −→ emptyTypeEnv

  • emptyStmt −→ emptyEnv
  • emptyStmt −→ emptyTypeEnv
  • constStmt (s1 , s2) −→ appendEnv (s1 , s2)
  • constStmt (s1 , s2) −→ appendTypeEnv (s1 , s2)
  • constStmt (s1 , s2) −→ appendList (s1 , s2)
  • constStmt (s1 , s2) −→ INH
  • constStmt (s1 , s2) −→ appendEnv (s1 , INH)
  • constStmt (s1 , s2) −→ appendTypeEnv (s1 , INH)

  • typeDcl (te , id) −→ emptyEnv
  • typeDcl (te , id) −→ consTypeEnv (id , te , emptyTypeEnv)
  • varDclType (te , id) −→ consEnvType (id , te , emptyEnv)
  • varDclType (te , id) −→ emptyEnv
  • varDclId (tid , id) −→ consEnvId (id , tid , emptyEnv)
  • varDclId (tid , id) −→ emptyEnv
  • assign (id , e) −→ emptyEnv
  • assign (id , e) −→ emptyTypeEnv
  • assign (id , e) −→ INH

  • varRef (id) −→ INH
  • varRef (id) −→ emptyList
  • varRef (id) −→ mkList (appendString (stringConstant , id))

  • boolType −→ boolTypeRep
  • intType −→ intTypeRep

  • consEnvType (id , t , rest) −→ appendEnv (singleEnv (id , t) , rest)
  • consEnvId (id , tid , rest) −→ INH
  • consEnvId (id , tid , rest) −→ consEnvType (id , INH , rest)
  • singleEnv (id , t) −→ t
  • singleEnv (id , t) −→ error (appendString (stringConstant , INH))
  • emptyEnv −→ error (appendString (stringConstant , INH))
```
• appendEnv (e1, e2) → e1
• appendEnv (e1, e2) → e2
• appendEnv (e1, e2) → INH

• consTypeEnv(tid, t, rest) → appendTypeEnv(singleTypeEnv(tid, t), rest)
• singleTypeEnv(tid, t) → t
• singleTypeEnv(tid, t) → error(appendString(stringConstant, INH))
• emptyTypeEnv → error(appendString(stringConstant, INH))
• appendTypeEnv(e1, e2) → e1
• appendTypeEnv(e1, e2) → e2
• appendTypeEnv(e1, e2) → INH

Consider the tree of locals on page 95 for the program on page 86 in grammar $G_2$. Consider the tree creation sequence defined by the path $T_0, T_1, T_2, T_4$ in this tree of locals. The trees correspond to the program syntax tree, the environment of the variable reference in the assignment statement, the type environment of this environment, and an appendTypeEnv tree created by the consTypeEnv sub-tree in this type environment.
Since the rules corresponding to the first and second trees contain the INH placeholder term, these steps cannot be modeled using the rewrite rules. The third tree creation step does not access inherited attributes, so this step can be modeled using the rewrite rules.

Since we can longer derive each local from its parent's sub-tree, Lemma \( \omega \) no longer holds. Therefore, we can no longer construct a rewrite sequence corresponding to each tree creation sequence, and so Lemma \( \rho \) no longer holds.

4.5.2 Bounding the Number of Inherited Accesses in Tree Creation Sequences

We present an analysis that checks the grammar for restrictions that ensure that the number of inherited accesses in a tree creation sequence is bounded. We then show that rewrite rules model model tree creation sub-sequences in which there are no inherited attributes. Thus if the rewrite rules terminate and the grammar satisfies the restrictions that ensure finite inherited accesses, then there are no infinite tree creation sequences and hence evaluation terminates.

The restrictions ensure that there is a function \( \mathcal{L} \) that maps the grammar’s non-terminals to a finite total order \( \text{leq} \) such that the following conditions are satisfied:

1. If there is a production with LHS \( X \) and RHS containing \( Y \) then \( \mathcal{L}(X) \leq \mathcal{L}(Y) \).
   i.e., non-terminals are non-decreasing from the root of a tree to its leaves.
2. If a synthesized attribute of type \( Y \) occurs on \( X \) then \( \mathcal{L}(X) \leq \mathcal{L}(Y) \).
3. If a local attribute of type \( Y \) occurs on a production with LHS \( X \) then \( \mathcal{L}(X) \leq \mathcal{L}(Y) \).
   i.e., the root non-terminal is non-decreasing during tree creation.
4. If an inherited attribute of type \( Y \) occurs on \( X \) then \( \mathcal{L}(X) < \mathcal{L}(Y) \).
   i.e., inherited occurs-on declarations are non-circular.
An algorithm for $\mathcal{L}$ is given in Section 4.5.4. It assigns non-negative sizes to the non-terminals based on how higher-order values are created in the grammar. The non-terminals in grammar $G_2$ defined on page 86 are ordered as follows:

<table>
<thead>
<tr>
<th>$X$</th>
<th>$\mathcal{L}(X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>0</td>
</tr>
<tr>
<td>Stmt</td>
<td>0</td>
</tr>
<tr>
<td>Expr</td>
<td>0</td>
</tr>
<tr>
<td>Type</td>
<td>0</td>
</tr>
<tr>
<td>Env</td>
<td>1</td>
</tr>
<tr>
<td>TypeEnv</td>
<td>2</td>
</tr>
<tr>
<td>TypeRep</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 4.10 shows a grammar whose non-terminals cannot be ordered as described in Section 4.5.2. The grammar defines a simple expression language with a `let` clause. A program in this grammar is shown below. The definitions in the `let` clauses are collected into an environment, which is used to perform look-ups in the `idRef` production.

```
let
 y = let
    x = 1
   in
    x + 3
 in
 y + 5
```

The non-terminals cannot be ordered as desired because the `Expr` non-terminal is decorated with an inherited attribute of non-terminal type `Env`, one of whose productions (`bindingEnv`) contains `Expr` in its right-hand side. This means that conditions 3 and 4 above cannot both be satisfied.

If the grammar non-terminals can be ordered and satisfy the properties above, we can order the trees in a tree creation sequence, by comparing their root non-terminals. If we order trees based on root non-terminals, every tree created as a local is at least as large as its parent.
**start** nonterminal Expr;
nonterminal Defs, Env;

synthesized attribute def :: Env occurs on Defs;
synthesized attribute value :: Integer occurs on Expr, Env;
synthesized attribute found :: Boolean occurs on Env;

- *Env occurs on Expr as an inherited attribute, and so Expr < Env.*
inherited attribute env :: Env occurs on Expr;
inherited attribute lookFor :: String occurs on Env;

concrete production emptyDefs d::Defs ::= { d.defs = emptyEnv (); }

concrete production consDefs d:Defs ::= id::Id_t '=' e:Expr ';' ds::Defs { d.defs = bindingEnv (id, e, ds.defs); }

concrete production let e::Expr ::= 'let' ds::Defs 'in' x::Expr { e.value = x.value; x.env = ds.defs; }

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 -1; }

concrete production intConst e::Expr ::= i::IntConst_t {e.value = mkInt(i.lexeme);}

concrete production add e::Expr ::= l::Expr '+' r::Expr { e.value = l.value + r.value; l.env = e.env; r.env = e.env; }

abstract production emptyEnv e::Env ::= { e.found = false; }

- *Expr occurs as a child of Env, and so Env <= Expr.*
abstract production bindingEnv e::Env ::= id::Id_t x::Expr rest::Env { e.found = id.lexeme == e.lookFor; e.value = if id.lexeme == e.lookFor then x.value else rest.value rest.lookFor = e.lookFor; }

Figure 4.10: $G_4$: a grammar whose non-terminals cannot be ordered as described in Section 4.5.2.
**Lemma λ**: Every tree created as a local is at least as large as its parent tree. Further, if the local is set to the value of an inherited attribute, then it is larger than its parent tree.

A proof for **Lemma λ** is given in Section A.5.2. Thus in any path in the tree of locals, the root non-terminals are non-decreasing, and in any section of the path in which root non-terminals do not increase, no locals are set to the values of inherited attributes. For example, here is the tree of locals from page 95 in which each tree has a value, based on the sizes assigned to the grammar’s non-terminals. Thus each node specifies the tree, its size, and the node of its parent on which it was created. In the tree creation sequence $T_0, T_1, T_2, T_4$, we have $T_0 < T_1 < T_2 \leq T_4$.

There is obviously no infinite increasing sequence of non-terminals. Therefore, there can be no tree creation sequence with an infinite number of steps where a larger tree is created, using inherited attributes. For any infinite tree creation sequence, there is a sub-sequence in which the trees are of the same size, in which there are no inherited attributes.

**Definition**: A *constant tree creation sequence* is a tree creation sequence in which the non-terminals at the roots of the trees are of the same size.
**Lemma** $\kappa$: If the required partial ordering exists for the grammar non-terminals, then for any infinite tree creation sequence, there exists a constant infinite tree creation sequence.

A proof for **Lemma** $\kappa$ is given in Section A.5.3.

### 4.5.3 Constant Tree Creation Sequences Can Be Modeled by the Rules

We can show that the rules do model the tree creation steps in a constant tree creation sequence. We assume that the roots of these trees are of the same size as the non-terminal $K$. Since the rewrite terms represent inherited accesses with the placeholder $\text{INH}$, we can no longer generate the actual local trees using the rules. Rather we generate "pruned" versions of the trees, in which some sub-trees are replaced by $\text{INH}$. The root symbols of the $\text{INH}$ sub-trees are greater than $K$. In effect, we ignore those nodes in the trees whose symbols are greater than $K$. 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.

Thus the rules can generate a pruned version of a local tree from any pruned version of its parent sub-tree. As in the case without higher-order inherited attributes, the rewrite sequence for each local is constructed in stages. For a particular constant tree-creation step, the first term of the sequence is a pruned partially evaluated term in which conditions are evaluated. Further, if the local is of the same size as $K$, then this first term will not be $\text{INH}$ and can be rewritten. The rest of the sequence rewrites the pruned sub-trees of this first term, on which synthesized instances are defined, to pruned versions of their evaluated values. We can thus generate the rewrite sequence for any higher-order evaluation step.

**Lemma** $\omega^{\text{INH}}$: Given a tree creation sequence (corresponding to a path in the tree of locals), a pruned version of every non-initial local tree can be derived via a non-empty rewrite rule sequence, from any pruned version of its parent’s sub-tree. Further, if the local’s root non-terminal is of the same size as its predecessor tree,
then the derived pruned term is not INH.

**Lemma** $\omega^{INH}$ is proved in Section A.5.4. Given a constant tree creation sequence, the sub-sequence that generate the pruned versions of each tree can be linked to generate a larger sequence that is at least as long as constant tree creation sequence.

**Lemma** $\rho^{INH}$: Given a constant tree creation sequence (corresponding to a path in the tree of locals), we can construct a rewrite sequence of at least the same length.

A formal proof for **Lemma** $\rho^{INH}$ is given in Section A.5.5. So if the rules terminate, then all constant tree creation sequences terminate Ordering non-terminals in this way therefore 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.

**Theorem** $II^{INH}$: If the rewrite rules are terminating and the non-terminals can be ordered as desired, there is no infinite constant tree creation sequence, and hence no improper evaluation sequence.

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. A formal proof for **Theorem** $II^{INH}$ is given in Section A.5.5.

### 4.5.4 A Procedure to Construct the Desired Ordering to Limit Inherited Accesses

In Figures 4.11 and 4.12, we present a procedure that, for a given grammar attempts to assign its non-terminals “levels” that satisfy the conditions in Section 4.5.2. The
1. All levels are initialized to -1.
   \[ \forall X \text{ SET level}(X) \text{ TO } -1. \]

2. \( NT_{INH} \) is set to all the non-terminals that are decorated by non-terminal inherited attributes or that are the types of inherited attributes.
   \[ \text{SET } NT_{INH} \text{ TO } \{X \mid X \in NT, \exists a_I \in A_I, \text{type}_a(a_I) = X\} \cup \{X \mid X \in NT, \exists a_I \in A_I, a_I@X\} \]

3. In phase 1, non-terminals are assigned increasing levels based on how they are decorated by or decorate as inherited attributes.
   \[ \text{SET currentLevel TO 0.} \]

4. \textbf{WHILE} \( NT_{INH} \neq \{\} \)
   
   \( (a) \) Level are assigned to the smallest non-terminals that are not the types of any remaining inherited attributes.
   \[ \text{SET currentSet TO} \{X \mid X \in NT_{INH}, (! \exists a_I \in A_I, Y \in NT_{INH}, a_I@Y, \text{type}_a(a_I) = X)\} \]

   \( (b) \) The analysis fails if there is a cycle among the inherited attributes.
   \[ \text{IF currentSet = \{\} AND NT_{INH} \neq \{\} \text{ RETURN FAILURE.} } \]

   \( (c) \) \( \forall X \in \text{currentSet} \)
   
   \( i. \) \text{SET level}(X) \text{ TO currentLevel.} \)
   \( ii. \) \text{SET } NT_{INH} \text{ TO } NT_{INH} \setminus \{X\}. \)

   \( (d) \) \text{SET currentLevel TO currentLevel + 1.} \)

Figure 4.11: Algorithm to assign non-terminals values that satisfy the conditions in Section 4.5.2, part 1 of 2.
5. – In phase 2, unassigned non-terminals are assigned decreasing levels based on
production signatures, and synthesized and local attributes.
– Levels are assigned repeatedly until a fix-point is reached.
SET currentLevel TO currentLevel – 1.

6. WHILE currentLevel > 0
   (a) SET changed TO true.
   (b) WHILE changed
      i. SET changed TO false.
      ii. SET currentSet TO
          \{Y \mid Y \in NT, (\exists X \in NT, level(X) = currentLevel AND
          (\exists a_S \in A_S, a_S \in X, type(a_S) = Y) OR
          (\exists p \in P, X = lhs(p), Y \in rhs(p)) OR
          (\exists p \in P, 1 \in L, 1 \in locals(p), X = lhs(p), Y = type_1(l)))\}
      iii. \forall X \in currentSet
          A. _IF_ level(X) == -1
              SET level(X) TO currentLevel.
              SET changed TO true.
          B. _The analysis fails if the orderings in the two phases are inconsistent._
              _IF_ level(X) < currentLevel _RETURN_ FAILURE.
   (c) SET currentLevel TO currentLevel – 1.

7. – All remaining non-terminals are assigned the lowest level.
\forall X \in NT . level(X) = -1 SET level(X) TO 0.

8. _RETURN_ SUCCESS.

Figure 4.12: Algorithm to assign non-terminals values that satisfy the conditions in
Section 4.5.2, part 2 of 2.
procedure can be used to show termination of the Ablej grammar in conjunction with its rules. Non-terminals are assigned their levels iteratively in two phases.

In phase 1, a subset of non-terminals is assigned levels based on inherited attributes so that an inherited attribute non-terminal is larger than the non-terminal it decorates. In each iteration, a working set is constructed of all unassigned non-terminals that are the types of inherited attributes, but do not themselves have any inherited occurrences. These are assigned the current level. If these non-terminals cannot be partially ordered because the inherited attribute relation is cyclical, the analysis fails.

In phase 2, unassigned non-terminals are assigned decreasing levels starting from the highest level assigned in phase 1. In this phase, levels are assigned based on production signatures and synthesized or local attributes so that a non-terminal is never larger than its local or synthesized attribute, or a child non-terminal. Since each assignment of a level can create further ordering constraints, the assignment of a particular level in phrase 2 is repeated until no assignments take place.

The analysis fails if the non-terminals cannot be assigned levels consistently in both phase 1 and phase 2. This would be the case for example if a non-terminal X decorates Y as an inherited attribute, but is decorated by Y as a synthesized attribute. Any non-terminals left unassigned at the end of phase 2 are assigned the lowest possible level and the procedure exits successfully.

We now look at an example of a successful assignment of levels. Figure 4.13 shows the phases of the analysis when it is run on the grammar $G_2$ defined in Figures 4.5, 4.6 and 4.7. We next look at an example in which the analysis fails. Figure 4.14 shows the phases of the analysis when it is run on the grammar $G_4$ defined in Figure 4.10.
Phase 1

Stmt < Env (inh), Expr < Env (inh), Stmt < TypeEnv (inh),
Expr < TypeEnv (inh), Env < TypeEnv (inh)

- currentLevel = 0, NT_{INH} = \{Stmt, Expr, Env, TypeEnv\},
currentSet = \{Stmt, Expr\}
- currentLevel = 1, NT_{INH} = \{Env, TypeEnv\}, currentSet = \{Env\}
- currentLevel = 2, NT_{INH} = \{TypeEnv\}, currentSet = \{TypeEnv\}
- currentLevel = 3, NT_{INH} = \{\}

Phase 2

- currentLevel = 2,
  non-terminals at current level = \{TypeEnv\},
currentSet = \{TypeEnv, TypeRep\} as
  TypeEnv \leq TypeRep (syn, prod), TypeEnv \leq TypeEnv (prod)

- currentLevel = 1,
  non-terminals at current level = \{Env\},
currentSet = \{TypeRep, TypeEnv, Env\} as
  Env \leq TypeRep (syn, prod), Env \leq TypeEnv (local), Env \leq Env (prod)

- currentLevel = 0,
  non-terminals at current level = \{Stmt, Expr\},
currentSet = \{Env, TypeEnv, Stmt, Type, Expr\} as
  Stmt \leq Env (syn), Stmt \leq TypeEnv (syn), Stmt \leq Stmt (prod),
  Stmt \leq Type (prod), Stmt \leq Expr (prod), Expr \leq Expr (prod)

Remaining non-terminals assigned 0: \{Root\}

SUCCESS

Figure 4.13: The phases of the analysis when it is run on the grammar \(G_2\) defined in Figures 4.5, 4.6 and 4.7
Phase 1

- Expr < Env (inh)
  - currentLevel = 0, $NT_{INH} = \{\text{Expr, Env}\}$, currentSet = $\{\text{Expr}\}$
  - currentLevel = 1, $NT_{INH} = \{\text{Env}\}$, currentSet = $\{\text{Env}\}$
  - currentLevel = 2, $NT_{INH} = \{}$

Phase 2

- currentLevel = 1,
  - non-terminals at current level = $\{\text{Env}\}$
  - currentSet = $\{\text{Expr, Env}\}$ as $\text{Env} \leq \text{Expr (prod)}$, $\text{Env} \leq \text{Env (prod)}$
  - \text{FAILURE} as level(Expr) = 0

Figure 4.14: The phases of the analysis when it is run on the grammar $G_4$ defined in Figure 4.10.
Chapter 5

Related Work and Conclusion

5.1 Related Work

Early approaches to language extensibility such as embedded DSLs [28], traditional syntactic, hygienic and programmable [29] macro systems, provided limited facilities for semantic analyses on new constructs, and little domain-specific feedback to the programmer. Meta-object protocol systems [30] and modern macros [31] 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 dependencies.

The extensible Java tool JavaBorg [32] allows the addition of new concrete syntax for objects. It is based on the MetaBorg embedding tool [33] that uses scanner-less GLR parsers. It performs generative, optimizing transformations via destructive rewrites. The underlying Stratego / XT rewriting system [34] 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 [35] (which like Silver, is functional), LRC [36] and JastAdd implement domain-specific translations in different ways. JastAdd [37] is an object-oriented extensible compiler tool that has been used to build a
Java 1.5 extensible compiler. Extensions are specified as rewritable reference attribute grammars, where reference attribute values are pointers to other (decorated) nodes [23].

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. 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 [38] used a notion similar to forwarding in a non-attribute grammar setting.

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 inter-leaved with destructive tree rewriting is harder to intuit. Thus JastAdd includes no static analysis for rewrite termination and performs only run-time checks. The functional nature of attribution evaluation in Silver and the semantics of forwarding allow for more termination analysis 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 composition of extensions, 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, both in terms of their functionality and their ability to be composed. And unlike systems like Polyglot [39] which require the specification of the order of code transformations, Silver aims to shield users from detailed implementation-level knowledge when composing extensions.
Appendix A

Technical Material from Chapter 4
(The Termination Analysis)

A.1 Introduction and Preliminary Definitions

A.1.1 Chapter Outline

In this chapter we provide the technical background for the analysis presented in Chapter 4. We provide formal definitions for the concepts we described informally in that chapter. We provide formal proofs for the theorems and lemmas presented in that chapter. The material in this chapter is presented to correspond with the sequence in which the analysis was described in Chapter 4. It is structured as follows.

In the rest of Section A.1, we provide preliminary definitions and notational conventions.

- In Section A.1.2 on page 125, we define context-free grammars and attribute grammars.
- In Section A.1.3 on page 127, we define syntax nodes and trees.
- In Section A.1.4 on page 127, we define tree terms, the term representation of the structure of syntax trees. These are the terms on which we will perform rewrites.
- In Section A.1.5 on page 128, we provide formal definitions of attribute instances and attributions.
In Section A.1.6 on page 129, we formally define attribute evaluation sequences and states.

In Section A.1.7 on page 130, we give a list of notational conventions.

In Section A.2 on page 131, we give a formal description of the class of attribute grammars described informally in Section 4.2. We define the syntax of its attribute definitions and expressions, and then formally describe our notions of evaluation sequences and tree creation sequences for this grammar.

In Section A.2.1 on page 131, we give the syntax of the attribute definitions and expressions in the restricted class of attribute grammars.

In Section A.2.2 on page 131, we define the conditions for attribute definitions and expressions to be type-correct.

In Section A.2.3 on page 134, we define the functions that generate attribute evaluation steps and sequences for a syntax tree in this restricted class of grammars.

In Section A.3 we formally prove the claim presented in Section 4.3 that for any non-terminating evaluation sequence, we can construct an infinite path in the tree of locals, and thus construct an infinite tree creation sequence.

In Section A.3.1 on page 136, we prove Theorem I, which states that for an infinite evaluation sequence, there is an infinite tree creation sequence.

In Section A.3.2 on page 142, we briefly consider the subset of expressions that define the trees in tree creation steps, and describe the structure of inductive proofs on them. Such proofs are used in the analysis description.

In Section A.4 we formally describe the first part of the termination analysis, presented informally in Section 4.4, that analyses grammars without inherited attributes.

In Section A.4.1 on page 145, we define the rewrite rules we use to model tree creation, and describe how they are generated for a given grammar.

In Section A.4.2 on page 147, we prove Lemma ω which states that if we generate rules in this way, we can derive each local in a tree creation sequence from its parent sub-tree.
In Section A.4.3 on page 155, we prove Lemma $\rho$ which states that for a given tree creation sequence, there exists a rewrite sequence of at least the same length. This gives us Theorem $II$ which states that in the absence of inherited attributes, if the rules terminate then tree creation terminates, and thereby evaluation terminates.

In Section A.5 we formally describe the second part of the analysis, described informally in Section 4.5 to handle inherited attributes.

- In Section A.5.2 on page 158, we describe a set of restrictions on the non-terminals in the grammar, derived from how higher-order expressions are defined and attribute occurrence relations. These restrictions can be used to order trees so that in every tree creation sequence, each tree is at least as large as its predecessor, and further, if an inherited access is performed, then the evaluated tree is strictly larger than its predecessor.

- In Section A.5.3 on page 159, we prove Lemma $\kappa$ which states that if a grammar’s non-terminals satisfy the restrictions, then in any infinite tree creation sequence, there is an infinite sub-sequence in which the non-terminals at the roots are of the same size.

- In Section A.5.4 on page 160, we prove Lemma $\omega^{INH}$ which states that we can use rewrite rules to model the parts of each tree creation step that do not access inherited access.

- In Section A.5.5 on page 169, we prove Lemma $\rho^{INH}$, which states that for a constant tree creation sequence, there is a rewrite sequence of at least the same length. This gives us Theorem $II^{INH}$ which states that if a grammar’s non-terminals satisfy these restrictions, and the grammar’s rules terminate, then tree creation terminates, which implies that evaluation terminates.

In Section A.3.2 on page 144, we give a brief note on the structure of inductive proofs on the structure of attribute definition expressions. We use such proofs in Sections A.4 and A.5.

A.1.2 Context-Free Grammars and Attribute Grammars

We start with definitions of the basic sets and functions used in our definitions and proofs. The following sets and functions are used to define context-free grammars [40].
• \( NT \) is a set of non-terminals.
• \( T \) is a set of terminals.
• \( P \) is a set of productions.

\[
\text{lhs} : P \rightarrow NT \text{ returns a production’s left-hand side.}
\]
\[
\text{rhs} : P \rightarrow (NT \cup T)^* \text{ returns a production’s right-hand side.}
\]

• \( S \in NT \) is the grammar’s start symbol.

The following sets and functions are used to define higher-order attribute grammars [7, 9].

• \( PT \) is a set of primitive types including \texttt{boolean}.
• \( PV \) is a set of primitive values including \texttt{true} and \texttt{false} of \texttt{boolean} type.
• \( Fun \) is a set of primitive functions.

\[
\text{inTypes} : Fun \rightarrow (PT)^* \text{ returns a function’s input types.}
\]
\[
\text{outType} : Fun \rightarrow PT \text{ returns a function’s output type.}
\]
\[
\text{applyFun} : Fun \rightarrow (PV)^* \rightarrow PV \text{ returns the result of applying a function to a list of values.}
\]

• \( A_S \) and \( A_I \) are sets of synthesized and inherited attributes respectively.
  
  \[
  \text{type}_a : (A_S \cup A_I) \rightarrow (PT \cup NT \cup T) \text{ returns an attribute’s type.}
  \]

• \( \subseteq (A_S \cup A_I) \times NT \) is a relation that specifies which attributes decorate which non-terminals.
• \( L \) is a set of higher-order local attributes.

\[
\text{type}_l : L \rightarrow NT \text{ returns a local attribute’s type.}
\]
\[
\text{locals} : P \rightarrow 2^L \text{ returns a production’s local attributes.}
\]

\( PT, PV \) and \( Fun \) give the primitive types, values and functions that will be used in attribute definition expressions. \( A_S, A_I \) and \( L \) give the typed synthesized, inherited and local attributes, respectively of the grammar. Finally \( \subseteq \) specifies the “occurs-on” relation between attributes and non-terminals.

Note that while synthesized and inherited attributes may be primitive or tree-valued,
local attributes are always non-terminal typed, a restriction explained below. Figure A.2
gives the values of these sets for a sample higher-order attribute grammar.

A.1.3 Syntax Nodes and Trees

We assume a set \( N \) of distinct identifiable syntax tree nodes. We have the following
functions on syntax tree nodes:

- \( \text{children} : N \rightarrow (N)^* \) returns a non-terminal node’s children as a list of nodes.
- \( \text{child} : N \rightarrow \text{INTEGER} \rightarrow N \) returns a node’s child node, given the child’s
  index.
  \[
  \text{child}(n, i) = n_i \text{ where } \text{children}(n) = n_1, \ldots, n_p
  \]
- \( \text{prod} : N \rightarrow P \) returns a node’s production.
- \( \text{symbol} : N \rightarrow \text{NT} \cup T \) returns a node’s symbol.

If \( \text{symbol}(n) \in \text{NT} \) and \( \text{children}(n) = n_1, \ldots, n_q \) then \( \text{lhs}(\text{prod}(n)) = \text{symbol}(n) \) and
\( \text{rhs}(\text{prod}(n)) = \text{symbol}(n_1), \ldots, \text{symbol}(n_q) \).

We will not define a separate type for syntax trees to avoid confusion with the notion
of tree terms defined below. We will instead refer to syntax trees by their root
nodes, as for example in the function \( \text{nodes} \):

\[
\text{nodes} : N \rightarrow 2^N \text{ returns the set of nodes in a syntax tree, given its root node.}
\]

\[
\text{nodes}(n) = \begin{cases} 
\{ n \} & \text{if } \text{symbol}(n) \notin \text{NT} \\
\{ n \} \cup \text{nodes}(n_1) \cup \ldots \cup \text{nodes}(n_q) & \text{if } \text{symbol}(n) \in \text{NT,} \\
& \quad \text{children}(n) = n_1, \ldots, n_q 
\end{cases}
\]

A.1.4 Tree Terms

We define an algebraic datatype \( \text{Term} \) that encapsulates the tree structure of a syntax
tree. These tree terms are the terms on which we will perform rewriting to abstract
higher-order attribution.

\( \text{Term} ::= \)
Also, as described below, the process of attribute evaluation distinguishes between higher-order values that are tree terms, and those that are full-fledged syntax trees with attribute instances on their nodes. We define the following functions to perform conversion between tree terms and syntax trees:

\[
\text{term}: N \rightarrow \text{Term} \quad \text{return the tree term for the sub-tree rooted at a given node.}
\]

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

\[
\text{newTree}: \text{Term} \rightarrow N \quad \text{given a tree term, returns the root of a syntax tree constructed with appropriate symbols and productions on its nodes.}
\]

### A.1.5 Attribute Instances and Attributions

During the process of attribute evaluation, each node of the syntax tree is associated with a set of attribute instances. The attribute grammar specifies how the values of these attribute instances are to be evaluated. We define the type \text{Instance} of attribute instances. Each attribute instance is defined by its node and attribute.

\[
\text{Instance} \equiv N \times (A_S \cup A_I \cup L)
\]

We will write \( n\#a \) for an instance \( \langle n, a \rangle \). Each syntax tree node has an attribute instance for each synthesized and inherited attribute that occurs on its symbol, and for each local attribute associated with its production. The function \text{instances} returns a tree’s attribute instances:

\[
\text{instances}: N \rightarrow 2^{\text{Instance}} \quad \text{returns all the attribute instances in a tree given its root.}
\]
instances(n) =

\[
\begin{cases}
\{n\#\text{lexeme}\} & \text{if } \text{symbol}(n) \in T \\
\{n\#a|a@\text{symbol}(n)\} \cup \{n\#1|l \in \text{locals}(q)\} & \text{if } \text{symbol}(n) \in NT, \text{prod}(n) = q, \\
\cup \text{instances}_n(n_1) \cup \ldots \cup \text{instances}_n(n_{n_q}) & \text{children}(n) = n_1, \ldots, n_{n_q}
\end{cases}
\]

An attribution is a partial map that associates values to the attribute instances of syntax trees. These values must be of the right type and be consistent with the attribute’s definition. We define Attribution to be the type of attributions.

Attribution ≡ Instance → (PV ∪ Term ∪ N ∪ {⊥})

A.1.6 Evaluation States and Sequences

We introduce a type State to represent the evaluation state at a particular point during the process of attribute evaluation. 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 in the trees. For clarity, we will represent a state by its root nodes and attribution. We assume that other nodes, and the edges between them are available in each state.

State ≡ 2^N × Attribution

The evaluation of a given tree can be formally described as a sequence of evaluation steps generated using the following functions:

- initState : Term → State returns the initial evaluation state for a given tree term
- nextState : State → Instance → State, given an evaluation state and a specific evaluable attribute instance returns the state that results from evaluating that attribute instance.

The initial state consists of a single syntax tree, and an attribution in which all the tree’s attribute instances are undefined.
\[\text{initState}(t) = \langle \{ n \}, \{ n'\#a' \mapsto \bot \mid n'\#a' \in \text{instances}(n) \} \rangle \text{ where } n = \text{newTree}(t)\]

In each later step an undefined attribute instance on one of the trees is selected and evaluated. We give the definition of the \text{nextState} function in Figure A.6 after we have described the syntax of our restricted class of attribute grammars.

An evaluation sequence for a tree \( t_0 \) is therefore given by a sequence
\[\langle T_0, \Gamma_0 \rangle, \langle T_1, \Gamma_1 \rangle, \langle T_2, \Gamma_2 \rangle, ... \text{ where } \langle T_0, \Gamma_0 \rangle = \text{initState}(t_0) \text{ and } \langle T_{i+1}, \Gamma_{i+1} \rangle = \text{nextState}(\langle T_i, \Gamma_i \rangle, n_i\#a_i)\]

Evaluation terminates if there are no undefined attribute instances that can be evaluated. Since there may be multiple attribute instances that can be evaluated in a given state, \text{initState} and \text{nextState} can be used to define multiple evaluation sequences, based on which evaluable attribute instance is selected for evaluation in each state.

### A.1.7 Notational Conventions

We use the following variables:

- \( NT \) is a set of non-terminals.
- \( X, Y, S, ... \in NT \) are non-terminals.
- \( T \) is a set of terminals.
- \( c \in T \) is a terminal symbol.
- \( P \) is a set of productions.
- \( p, q \in P \) are productions.
- \( Fun \) is a set of primitive functions.
- \( f \in Fun \) is a function.
- \( AS \) is a set of synthesized attributes.
- \( a_s \in AS \) is a synthesized attribute.
- \( AI \) is a set of inherited attributes.
- \( a_i \in AI \) is an inherited attribute.
- \( L \) is a set of higher-order local attributes.
- \( l \in L \) is a local attribute.
A.2 Section 4.2 (A Restricted Class of Attribute Grammars)

A.2.1 Attribute Definitions and Expressions

We start by describing the class of higher-order attribute grammars handled by our termination analysis. As explained in Section 4.2.1, the restrictions we place on the definitions in the grammar allow us to reduce the termination problem, to that of disproving infinite local tree creation sequences. The attribute grammar associates each production of the context-free grammar with a set of attribute definitions of type Defn.

\[
defs : P \rightarrow 2^{\text{Defn}} \text{ returns a production's attribute definitions.}
\]

Each definition specifies the function that defines the value of a synthesized, inherited or local instance on syntax tree nodes. Definitions are of the form \( \text{occurrence} = \text{expression} \) and have the syntax defined in Figure A.1.

The right-hand sides of definitions are attribute defining expressions of type \( \text{Expr} \). Figure A.1 gives the syntax of these expressions in our restricted class of attribute grammars. Note that all tree terms are valid expressions, i.e., \( \text{Term} \subseteq \text{Expr} \). Figure A.2 shows an example of the definition of an attribute grammar in this format. A Silver version of this grammar is given in Figure 4.10.

A.2.2 Type-Correct Definitions and Expressions

In a valid attribute grammar, attribute definitions must be correctly typed. The types of expressions on the right-hand side of attribute definitions are defined as shown in Figure A.3. These can be used to define valid definitions and expressions as shown in Figure A.4 and Figure A.5. Note that the static type and validity of an expression or definition are determined for a specific production, since signature variables and local references must be resolved for particular productions. In the termination analysis, we assume that we are analysing only valid grammars, i.e., we have

- \( e, e_C, e_T, e_E, e_L, e_1, \ldots \in \text{Expr} \) are attribute definition expressions.
**Defn** is the type of attribute definitions.

\[
\text{Defn ::=} \\
\quad \#0.a_S = \text{Expr} \quad \text{(syn. occ. on parent)} \\
\mid \quad \#i.a_I = \text{Expr} \quad \text{(inh. occ. on child)} \\
\mid \quad l = \text{Expr} \quad \text{(local occ.)} \\
\mid \quad l.a_I = \text{Expr} \quad \text{(inh. occ. on local)}
\]

**Expr** is the type of expressions in the RHS of attribute definitions.

\[
\text{Expr ::=} \\
\quad \#i \quad \text{(child tree)} \\
\mid \quad \#0.a_I \quad \text{(inh. occ. on parent)} \\
\mid \quad \#i.a_S \quad \text{(syn. occ. on child)} \\
\mid \quad l.a_S \quad \text{(syn. occ. on local)} \\
\mid \quad q(\text{Expr}_1, \ldots, \text{Expr}_n) \quad \text{(tree creation)} \\
\mid \quad c \quad \text{(terminal symbol)} \\
\mid \quad f(\text{Expr}_1, \ldots, \text{Expr}_n) \quad \text{(function call)} \\
\mid \quad \text{if Expr}_C \text{ then Expr}_T \text{ else Expr}_E \quad \text{(conditional expr.)}
\]

Figure A.1: The syntax of attribute grammar definitions and expressions.
- $NT = \{\text{Expr}, \text{Defs}, \text{Env}\}$
- $T = \{'\};', 'Id_t', '='; ', 'let', 'in', \text{IntConst}_t, '+'\}$
- $S = \text{Expr}$

- $PT = \{\text{integer}, \text{boolean}, \text{string}\}$
- $PV = \{\text{true}, \text{false}, -1\}$
- $A_S = \{\text{defs} :: \text{Env}, \text{value} :: \text{integer}, \text{found} :: \text{boolean}\}$
- $A_I = \{\text{env} :: \text{Env}, \text{lookFor} :: \text{string}\}$

- $\& = \{\langle \text{defs} :: \text{Defs} \rangle, \langle \text{value} :: \text{Expr} \rangle, \langle \text{value} :: \text{Env} \rangle, \langle \text{env} :: \text{Expr} \rangle, \langle \text{lookFor} :: \text{Env} \rangle, \langle \text{found} :: \text{Env} \rangle\}$

- $\text{Fun} = \{\text{mkInt} :: \text{string} \rightarrow \text{integer}, + :: \text{integer} \rightarrow \text{integer} \rightarrow \text{integer}, == :: \text{string} \rightarrow \text{string} \rightarrow \text{boolean}\}$

- $L = \{\text{lenv} :: \text{Env}\}$
- $\text{locals(idRef)} = \{\text{lenv}\}$

- $P = \{\text{emptyDefs}, \text{consDefs}, \text{let}, \text{idRef}, \text{intConst}, \text{add}, \text{emptyEnv}, \text{bindingEnv}\}$

emptyDefs ::Defs $\rightarrow$ ';'
  #0.defs = emptyEnv()

consDefs ::Defs $\rightarrow$ Id_t '=' Expr ';'Defs
  #0.defs = bindingEnv(#1, #3, #5.defs)

let :: Expr $\rightarrow$ 'let'Defs 'in'Expr
  #0.value = #4.value
  #4.env = #2.defs

idRef :: Expr $\rightarrow$ Id_t
  lenv = #0.env
  lenv.lookFor = #1.lexeme
  #0.value = if lenv.found then lenv.value else -1

intConst :: Expr $\rightarrow$ IntConst_t
  #0.value = mkInt(#1.lexeme)

add :: Expr $\rightarrow$ Expr '+' Expr
  #0.value = #1.value + #3.value
  #1.env = #0.env
  #3.env = #0.env

emptyEnv :: Env $\rightarrow$
  #0.found = false

bindingEnv :: Env $\rightarrow$ Id_t Expr Env
  #0.found = #1.lexeme == #0.lookFor
  #0.value = if #1.lexeme == #0.lookFor then #2.value else #3.value
  #3.lookFor = #0.lookFor

Figure A.2: An example of the definition of an attribute grammar.
type \( e \) returns an expression's type.

\[
\text{type} \ e : P \rightarrow \text{Expr} \rightarrow (PT \cup NT \cup T)
\]

For an expression \( e \) on a production \( p \), \( \text{type}_e(p, e) \) is defined as follows:

\[
\begin{align*}
\text{type}_e(p, \#i) &= \text{rhs}(p)^i & \text{(type of } i^{\text{th}} \text{ child)} \\
\text{type}_e(p, \#0.a_1) &= \text{type}_e(a_1) & \text{(type of inh. occ. on parent)} \\
\text{type}_e(p, \#i.a_5) &= \text{type}_e(a_5) & \text{(type of syn. occ. on } i^{\text{th}} \text{ child)} \\
\text{type}_e(p, .a_3) &= \text{type}_e(a_3) & \text{(type of syn. occ. on local)} \\
\text{type}_e(p, q(e_1, ..., e_n)) &= \text{lhs}(q) & \text{(production's LHS symbol)} \\
\text{type}_e(p, c) &= c & \text{(terminal symbol)} \\
\text{type}_e(p, f(e_1, ..., e_n)) &= \text{outType}(f) & \text{(function's output type)} \\
\text{type}_e(p, \text{if } e_C \text{ then } e_T \text{ else } e_E) &= \text{type}(e_T) & \text{(sub-expr. type)}
\end{align*}
\]

Figure A.3: Defining the type of attribute expressions.

\[
\forall p \in P \cdot \forall d \in \text{defs}(p) \cdot \text{isValidDefn}(p, d)
\]

### A.2.3 Defining the Steps in an Evaluation Sequence

For this class of attribute grammars, the steps of evaluation sequences are generated as shown in Figure A.6. The set of required attribute instances of an expression is given in Figure A.7. The set of attribute occurrences required for the evaluation of each instance can only be computed lazily at run-time, because of the semantics of conditions and local attributes. We say an attribute instance is evaluable in a state if it may be selected for evaluation, i.e. if

\[
\Gamma[n#a] = \perp \text{ and } \forall (n'#a') \in \text{dep}(n, e, \Gamma). \Gamma[n'#a'] \neq \perp
\]

where \( e \) is \( n#a \)'s defining expression.

The evaluated value of expressions is defined in Figure A.8. As shown in Figure A.8, the syntax of a higher-order expression does not indicate whether it will be evaluated to a tree term or syntax tree. Rather, as shown in Figure A.6, this is decided by the evaluation context, i.e., if a local instance is being evaluated. The set of possible
isValidExpr is true if a given expression is type-correct for a given production.

$$\text{isValidExpr} : P \rightarrow \text{Expr} \rightarrow \text{BOOL}$$

For an expression $e$ on a production $p$, $\text{isValidExpr}(p, e)$ is defined as follows:

$$1 \leq i \leq |\text{rhs}(p)|$$
$$\text{isValidExpr}(p, \#i)$$

(child tree)

$$a_1 \@ \text{lhs}(p)$$
$$\text{isValidExpr}(p, \#0.a_1)$$

(inh. occ. on parent)

$$1 \leq i \leq |\text{rhs}(p)|, a_8 \@ (\text{rhs}(p)^i)$$
$$\text{isValidExpr}(p, \#i.a_8)$$

(syn. occ. on child tree)

$$l \in \text{locals}(p), a_8 \@ \text{type}_1(l)$$
$$\text{isValidExpr}(p, \#l.a_8)$$

(syn. occ. on local tree)

$$\text{rhs}(q) = \text{type}_e(p, e_1), \ldots, \text{type}_e(p, e_n)$$
$$\text{isValidExpr}(p, q(e_1, \ldots, e_n))$$

(valid sub-tree types)

$$\text{isValidExpr}(p, c)$$

(terminal symbol)

$$\text{inTypes}(f) = \text{type}_e(p, e_1), \ldots, \text{type}_e(p, e_n)$$
$$\text{isValidExpr}(p, f(e_1, \ldots, e_n))$$

(valid argument types)

$$\text{type}_e(p, e_C) = \text{BOOL}, \text{type}_e(p, e_T) = \text{type}_e(p, e_E)$$
$$\text{isValidExpr}(p, \text{if } e_C \text{ then } e_T \text{ else } e_E)$$

(same sub-expr. type)

Figure A.4: Valid attribute definition expressions.
isValidDefn is true if a given definition is type-correct for a given production.

\[
\text{isValidDefn}: P \rightarrow \text{Defn} \rightarrow BOOL
\]

For a definition \(d\) on a production \(p\), \(\text{isValidDefn}(p, d)\) is defined as follows:

\[
\begin{align*}
\text{isValidDefn}(p, \#0.a_S = e) & \quad \text{(syn. occ.)} \\
\text{isValidDefn}(p, \#i.a_I = e) & \quad \text{(inh. occ. on child)} \\
\text{isValidDefn}(p, l = e) & \quad \text{(local def.)} \\
\text{isValidDefn}(p, \#i.a_I = e) & \quad \text{(inh. occ. on local)}
\end{align*}
\]

Figure A.5: Valid attribute definitions.

values assigned by an attribution includes tree nodes and the \(\bot\) value, in addition to the primitive values and tree terms returned by \(\text{eval}\).

A.3 Section 4.3 (Reducing the Problem to Disproving Infinite Tree Creation Sequences)

A.3.1 Reducing Infinite Evaluation to Infinite Tree Creation

As described in Section 4.3, we can reduce the problem of guaranteeing termination of attribution to that of disproving the existence of infinite tree creation sequences by showing the following:

- Any improper evaluation sequence for a non-circular grammar is infinite.
- Any infinite evaluation sequence contains an infinite number of local tree creation steps. For such a sequence, we can construct trees of locals with increasing numbers of nodes and edges.
**nextState**, given an evaluation state and an evaluable attribute instance, returns the state that results from evaluating that attribute instance.

\[
\text{nextState : State} \rightarrow \text{Instance} \rightarrow \text{State}
\]

For a state \( \langle T, \Gamma \rangle \) and evaluable attribute instance \( n\#a \in \text{instances}(t), t \in T \)

\text{nextState}(\langle T, \Gamma \rangle, n\#a) \) is defined as follows:

<table>
<thead>
<tr>
<th>defining node, expression</th>
<th>nextState(\langle T, \Gamma \rangle, n#a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a \in A_S ) ( #0.a = e ) \in \text{defs}(\text{prod}(n)) ) \hspace{1cm} (syn. occ. on child)</td>
<td>( \langle T, \Gamma[n#a \mapsto \text{eval}(n, e, \Gamma)] \rangle )</td>
</tr>
<tr>
<td>( a \in A_I ) ( n = \text{child}(n_p, i), ) \hspace{1cm} (inh. occ. on child) \hspace{1cm}</td>
<td>( \langle T, \Gamma[n#a \mapsto \text{eval}(n_p, e, \Gamma)] \rangle )</td>
</tr>
<tr>
<td>( a \in L ) ( a = e ) \in \text{defs}(\text{prod}(n_p)), ) \hspace{1cm} \hspace{1cm}</td>
<td>( \langle T \cup {t_L}, \Gamma[n#a \mapsto t_L] \cup {[n'#a' \mapsto \perp]</td>
</tr>
<tr>
<td>( n = \text{newTree} (\text{eval}(n, e, \Gamma)) ) \hspace{1cm} (new local tree with undef. inst.) \hspace{1cm}</td>
<td>( \langle T, \Gamma[n#a \mapsto \text{eval}(n_p, e, \Gamma)] \rangle )</td>
</tr>
<tr>
<td>( a \in A_I ) ( n = \Gamma[n_p#a], ) \hspace{1cm} (inh. occ. on local) \hspace{1cm}</td>
<td>( \langle T, \Gamma[n#a \mapsto \text{eval}(n_p, e, \Gamma)] \rangle )</td>
</tr>
</tbody>
</table>

Figure A.6: Defining the non-initial states in an evaluation sequence.
\texttt{dep} returns the set of attribute instances required to evaluate an expression on a given node during evaluation.

\[
\text{dep} : N \rightarrow \text{Expr} \rightarrow \text{Attribution} \rightarrow 2^{\text{Instance}}
\]

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{dep}(n, #i, \Gamma)</td>
<td>{ } (child tree)</td>
</tr>
<tr>
<td>\text{dep}(n, #0.a_1, \Gamma)</td>
<td>{n#a_1} (inh. inst. on parent)</td>
</tr>
<tr>
<td>\text{dep}(n, #i.a_S, \Gamma)</td>
<td>{\text{child}(n,i)#a_S} (syn. inst. on child)</td>
</tr>
</tbody>
</table>
| \text{dep}(n, l.a_S, \Gamma) | \begin{cases} 
\{n\#1\} & \text{if } \Gamma[n\#1] = \bot \\
\{(\Gamma[n\#1])\#a_S\} & \text{otherwise}
\end{cases} 
| (local instance if it’s undef., else the local syn. inst.) |
| \text{dep}(n, q(e_1, ..., e_n), \Gamma) | \bigcup_{i=1}^{n_q} \text{dep}(n, e_i, \Gamma) (req. inst. on all sub-exprs.) |
| \text{dep}(n, c, \Gamma) | \{ \} (terminal symbol) |
| \text{dep}(n, f(e_1, ..., e_n), \Gamma) | \bigcup_{i=1}^{n_f} \text{dep}(n, e_i, \Gamma) (req. inst. on all sub-exprs.) |
| \text{dep}(n, \text{if } e_C \text{ then } e_T \text{ else } e_E, \Gamma) | \begin{cases} 
\text{dep}(n, e_C, \Gamma) & \text{if } \text{dep}(n, e_C, \Gamma) \neq \bot \\
\text{dep}(n, e_T, \Gamma) & \text{if } \text{eval}(n, e_C, \Gamma) = \text{true} \\
\text{dep}(n, e_E, \Gamma) & \text{otherwise}
\end{cases} 
| (cond.‘s req. inst. if not evaluable, else req. inst. in right sub-exp.) |

Figure A.7: Defining the set of attribute instances required to evaluate an expression.
eval returns the value of an evaluable expression evaluated on a node for a given attribution.

\[
\text{eval: } N \rightarrow \text{Expr} \rightarrow \text{Attribution} \rightarrow (PV \cup \text{Term})
\]

For an expression \( e \) on a node \( n \), \( \text{eval}(n, e, \Gamma) \) is defined as follows:

\[
\begin{align*}
\text{eval}(n, \#i, \Gamma) &= \text{term}(\text{child}(n, i)) & (i^{\text{th}} \text{ child tree}) \\
\text{eval}(n, \#0.a_1, \Gamma) &= \Gamma[n\#a_1] & (\text{inh. inst. value on parent}) \\
\text{eval}(n, \#i.a_S, \Gamma) &= \Gamma[\text{child}(n, i)\#a_S] & (\text{syn. inst. value on child}) \\
\text{eval}(n, \_a_S, \Gamma) &= \Gamma[(\Gamma[n\#1])\#a_S] & (\text{syn. inst. value on local}) \\
\text{eval}(n, q(e_1, ..., e_nq), \Gamma) &= q(\text{eval}(n, e_1, \Gamma), ..., \text{eval}(n, e_nq, \Gamma)) & (\text{evaluated tree term}) \\
\text{eval}(n, c, \Gamma) &= c & (\text{terminal symbol}) \\
\text{eval}(n, f(e_1, ..., e_nf), \Gamma) &= \text{applyFun}(f, \text{eval}(n, e_1, \Gamma), ..., \text{eval}(n, e_nf, \Gamma)) & (\text{value of function call}) \\
\text{eval}(n, \text{if } e_C \text{ then } e_T \text{ else } e_E, \Gamma) &= \begin{cases} 
\text{eval}(n, e_T, \Gamma) & \text{if } \text{eval}(n, e_C, \Gamma) = \text{true} \\
\text{eval}(n, e_E, \Gamma) & \text{otherwise}
\end{cases} & (\text{evaluated sub-expr.})
\end{align*}
\]

Figure A.8: Defining the evaluated value of an attribute definition expression.
We can then construct an infinite path of locals from the original tree, in which each non-initial tree is created as a local on its predecessor.

**Theorem I**: For a tree in a non-circular grammar with terminating functions, if there is an improper evaluation sequence, then there is an infinite tree creation sequence.

**Proof**: Proof is by construction.

- Assume an improper evaluation sequence for a tree $t_0$ in a non-circular grammar with terminating functions, given by $\langle T_0, \Gamma_0 \rangle, \langle T_1, \Gamma_1 \rangle, \langle T_2, \Gamma_2 \rangle, \ldots$ where
  - $\langle T_0, \Gamma_0 \rangle = \text{initState}(t_0)$ and
  - $\langle T_{i+1}, \Gamma_{i+1} \rangle = \text{nextState}(\langle T_i, \Gamma_i \rangle, n_i \# a_i)$.

- Since the grammar is non-circular and the sequence is improper, there is an evaluable instance in every evaluation step. i.e., at every evaluation step $\langle T_i, \Gamma_i \rangle$, there exists $n_i \# a_i \in \text{instances}(t_i)$ for $t \in T_i$ where $\langle T_{i+1}, \Gamma_{i+1} \rangle = \text{nextState}(\langle T_i, \Gamma_i \rangle, n_i \# a_i)$.

- Thus the evaluation sequence is infinite.

- Since each tree in any state has a finite number of instances, and every step evaluates an instance, the infinite evaluation sequence has an infinite number of instance-adding local tree creation steps.

- Let $L_0, L_1, L_2, \ldots$ be the indices of all local-instance evaluating steps where $L_0 = 0$ and $L_0, L_1, L_2, \ldots$ is a sub-sequence of 0, 1, 2, ... We have an infinite sub-sequence of local tree creation steps $\langle T_{L_0}, \Gamma_{L_0} \rangle, \langle T_{L_1}, \Gamma_{L_1} \rangle, \langle T_{L_2}, \Gamma_{L_2} \rangle, \ldots$ where
  - $\mathcal{T}_{L_0} = \mathcal{T}_0$, $\mathcal{\Gamma}_{L_0} = \Gamma_0$ and
  - for $i > 0$
    - $\mathcal{T}_{L_i} = \mathcal{T}_{L_{i-1}} \cup \{ t_{L_i} \}$ and
    - $\mathcal{\Gamma}_{L_i} = \mathcal{\Gamma}_{L_{i-1}} \cup \{ \text{instances}(t_{L_i}) \}$

where
- \( t \in \mathcal{T}_{L_{i-1}} \),
- \( n_{L_i} \#1_{L_i} \in \text{instances}(t) \),
- \( (1_{L_i} = e_{L_i}) \in \text{defs(prod}(n_{L_i}) \)) and
- \( t_{L_i} = \text{newTree(eval}(n_{L_i}, e_{L_i}, \Gamma_{L_{i-1}})) \).

• Since this sequence includes all local-instance evaluating steps, there is no change in the set of locals between any two tree creation steps, i.e., between \( \mathcal{T}_{L_{i-1}} \) and \( \mathcal{T}_{L_{(i-1)}} \), and so we have \( \mathcal{T}_{L_{i-1}} = \mathcal{T}_{L_{(i-1)}} \).

For example, if exactly every fifth step in the evaluation sequence is a local instance evaluating step, then for the evaluation sequence \( \langle \mathcal{T}_0, \Gamma_0 \rangle, \langle \mathcal{T}_1, \Gamma_1 \rangle, \langle \mathcal{T}_2, \Gamma_2 \rangle, \ldots \) we have the local creation sequence

\[
\langle \mathcal{T}_0, \Gamma_0 \rangle, \langle \mathcal{T}_4, \Gamma_4 \rangle, \langle \mathcal{T}_9, \Gamma_9 \rangle, \ldots, \text{or } \langle \mathcal{T}_{L_0}, \Gamma_{L_0} \rangle, \langle \mathcal{T}_{L_1}, \Gamma_{L_1} \rangle, \langle \mathcal{T}_{L_2}, \Gamma_{L_2} \rangle, \ldots \text{ where } L_0 = 0, L_1 = 4, L_2 = 9, \ldots \text{ For } i = 2 \text{, we have } L_i = 9 \text{ and } \mathcal{T}_{L_{i-1}} = \mathcal{T}_8 = \mathcal{T}_{L_{(i-1)}} = \mathcal{T}_4 \).

Thus the infinite sub-sequence of local evaluation steps above can be defined by

\[
\langle \mathcal{T}_{L_0}, \Gamma_{L_0} \rangle, \langle \mathcal{T}_{L_1}, \Gamma_{L_1} \rangle, \langle \mathcal{T}_{L_2}, \Gamma_{L_2} \rangle, \ldots \text{ where } \\
\mathcal{T}_{L_0} = \mathcal{T}_0, \Gamma_{L_0} = \Gamma_0 \text{ and } \\
\text{ for } i > 0 \text{ we have } \\
\mathcal{T}_{L_i} = \mathcal{T}_{L_{(i-1)}} \cup \{t_{L_i}\} \text{ and } \\
\Gamma_{L_i} = \Gamma_{L_{i-1}}, \left[n_{L_i} \#1_{L_i} \mapsto t_{L_i}\right] \cup \left\{\left[n' \#a' \mapsto \bot\right] \mid n' \#a' \in \text{instances}(t_{L_i})\right\}
\]

where

- \( t \in \mathcal{T}_{L_{(i-1)}} \),
- \( n_{L_i} \#1_{L_i} \in \text{instances}(t) \),
- \( (1_{L_i} = e_{L_i}) \in \text{defs(prod}(n_{L_i}) \)) and
- \( t_{L_i} = \text{newTree(eval}(n_{L_i}, e_{L_i}, \Gamma_{L_{i-1}})) \).

• Since each local is evaluated exactly once, on a pre-existing tree, each set \( \mathcal{T}_{L_i} \) of trees can be organized into a tree of locals (TOL) in which each node has a finite number of children.
• There is thus an infinite sequence $T_{L_0}, T_{L_1}, T_{L_2}, \ldots$ of larger and larger TOLs in which each node in each TOL has a finite number of children.

• König’s Lemma states that an infinite tree in which each node has finite children, has an infinite path. It can be used to construct an infinite path of tree creation steps starting from the original tree, in which each non-initial tree is created as a local on its predecessor. We thus have an infinite tree creation sequence $\langle t_{k_0}, \Gamma_{k_0} \rangle, \langle t_{k_1}, \Gamma_{k_1}, n_{k_1}\#l_{k_1} \rangle, \langle t_{k_2}, \Gamma_{k_2}, n_{k_2}\#l_{k_2} \rangle, \ldots$ where $k_0, k_1, k_2, \ldots$ is a sub-sequence of $L_0, L_1, L_2, \ldots$.

\[ T_{k_0} = T_0, \Gamma_{k_0} = \Gamma_0 \]

for $i > 0$

\[ t_{k_i} = \text{newTree} (\text{eval}(n_{k_i}, e_{k_i}, \Gamma_{k_i-1})) \text{ and } \]

\[ \Gamma_{k_i} = \Gamma_{k_i-1}[n_{k_i}\#l_{k_i} \mapsto t_{k_i}] \cup \{ n'\#a' \mapsto \bot \mid n'\#a' \in \text{instances}(t_{k_i}) \} \]

where

- $n_{k_i}\#l_{k_i} \in \text{instances}(t_{k_{i-1}})$ and
- $(l_{k_i} = e_{k_i}) \in \text{defs}(\text{prod}(n_{k_i}))$.

• Thus for any improper evaluation sequence, there is an infinite tree creation sequence from the original tree, in which each non-initial tree is created as a local on its predecessor.

### A.3.2 Higher-Order Tree Creating Expressions

As a result of Theorem $I$, the rest of the analysis deals primarily with tree creation evaluation steps. These steps evaluate a particular category of attributes, with definitions of the form $1 = \text{Expr}$. The higher-order expressions on the right-hand sides of these definitions are a subset of $\text{Expr}$, the general set of expressions that define attributes. They are defined by the grammar shown in Figure A.9. 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
Defn is the type of attribute definitions.

Defn ::= l = Expr

Expr is the type of expressions in the RHS of attribute definitions, defined by the condition: type(e) ∈ NT ∪ T. Such expressions are given by the following grammar:

Expr ::= #i
| #0.aI                      (aI ∈ NT ∪ T)
| #i.as                     (as ∈ NT ∪ T)
| l.as                     (as ∈ NT ∪ T, local is defined)
| q(Expr1, ..., Exprnq)    
| c
| if ExprC then ExprT else ExprE   (ExprC is not higher-order)

dep returns the set of attribute instances required to evaluate an expression.

dep : N → Expr → Attribution → 2^Instance

For an expression evaluable higher-order e on a node n, dep(n, e, Γ) is defined as follows:

dep(n, #i, Γ) = { }
dep(n, #0.aI, Γ) = {n#aI}                  (aI ∈ NT ∪ T)
dep(n, #i.as, Γ) = {child(n,i)#as}          (as ∈ NT ∪ T)
dep(n, l.as, Γ) = {(Γ[n#1])#as}           (as ∈ NT ∪ T)

dep(n, q(e1, ..., enq), Γ) = \bigcup_{i=1}^{nq} dep(n, ei, Γ)
dep(n, c, Γ) = { }
dep(n, if eC then eT else eE, Γ)
= \begin{cases} 
  dep(n, eT, Γ) & \text{if eval}(n, eC, Γ) = \text{true} \\
  \text{false} & \text{otherwise} 
\end{cases}

Figure A.9: The set of expressions that are evaluated to local tree values, and their required instances.
of attributes that are of non-terminal or terminal types are included. If the grammar’s definitions and expressions are type correct then tree evaluated as a local has a valid structure.

**A Note on Inductive Proofs on Expressions** Some of the proofs in this chapter are inductive proofs on the structure of evaluable higher-order expression. In this section we briefly describe the structure of these proofs. An evaluable higher-order expression on a production is any valid expression on the RHS of any of its definitions such that all of its required instances are defined.

Some of the proofs assume that the property in question is satisfied by all required occurrences of a given higher-order evaluable expression. We can show that any evaluable higher-order expression is such that all its required instances are also higher-order, as shown in Figure A.9. This is obvious in every case except conditional expressions. As shown in Figure A.7, the set of required instances for a higher-order conditional expression may include non-higher order instances, viz., those present in its condition. However, if the conditional expression is evaluable, then the set of required instances is defined to be only the required instances of the appropriate sub-expression, which are all higher-order. Synthesized attribute accesses off locals are similarly defined differently if the expression is evaluable.

To show that a property \( P \) holds on all evaluable higher-order expressions on a production using a proof by induction, 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 its sub-expressions satisfy \( P \).

- For synthesized attributes on locals, we assume \( P \) holds on the local’s production and its valid evaluable higher-order expressions, as well as the local’s defining expression on the parent production.
• 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.

A.4 Section 4.4 (Modeling Tree Creation During Evaluation with Rewrite Rules)

A.4.1 Constructing Rewrite Rules to Model Tree Creation

In this section we use rewrite rules to model tree creation so that termination of the rewrite rules implies termination of all tree creation sequences. These rules rewrite terms that are the sub-trees on which the local is evaluated, to the terms that are the result of the tree creation step, viz., the local tree term. Each rule is associated with a specific production. It rewrites tree terms with the production at its root (the local’s parent sub-tree) to evaluated local tree terms. The type $\text{Rule}$ represents rules.

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

where $\text{RuleRHS}$ represents the the right-hand sides of these rewrite rules. These are the same as tree terms except they may refer to sub-terms on the left-hand sides via an index on the production signature.

$$\text{RuleRHS} ::= \#	ext{\text{\texttt{i}}} \quad (i^{\text{th}} \text{ term variable})$$
$$| \text{c} \quad (\text{terminal symbol})$$
$$| q(\text{RuleRHS}_1, ..., \text{RuleRHS}_n) \quad (\text{prod. symbol and sub-expres.})$$

Note that $\text{Term}$ is the set of tree term values while $\text{RuleRHS}$ and $\text{Expr}$ are expressions over these values. So we have $\text{Term} \subseteq \text{RuleRHS} \subseteq \text{Expr}$.

For a given grammar, we derive a set of rewrite rules based on the higher-order attribute definitions in its productions, as explained in Section 4.4.1.
ruleRHSs returns the right-hand sides of the rules that model the evaluation of a given higher-order expression.

\[
\text{ruleRHSs}: P \rightarrow \text{Expr} \rightarrow 2^{\text{RuleRHS}}
\]

For an expression \(e\) on a production \(p\) where \(\text{type}_e(e) \in (NT \cup T)\), \(\text{ruleRHSs}(p, e)\) is defined as follows:

\[
\begin{align*}
\text{ruleRHSs}(p, \#i) &= \{\#i\} \quad (i^{th} \text{ sub-term}) \\
\text{ruleRHSs}(p, \#i.a) &= \{\#i\} \quad (i^{th} \text{ sub-term}) \\
\text{ruleRHSs}(p, l.a) &= \text{ruleRHSs}(p, e_L) \quad (\text{derived from local’s defn.}) \\
&\quad \text{where } (l = e_L) \in \text{defs}(p)
\end{align*}
\]

\[
\begin{align*}
\text{ruleRHSs}(p, q(e_1, ..., e_n)) &= \{q(r_1, ..., r_n)\} \quad (\text{rule for all sub-rule combinations}) \\
&\quad | r_i \in \text{ruleRHSs}(p, e_i), 1 \leq i \leq n_q
\end{align*}
\]

\[
\begin{align*}
\text{ruleRHSs}(p, c) &= \{c\} \quad (\text{terminal symbol})
\end{align*}
\]

\[
\begin{align*}
\text{ruleRHSs}(p, \text{if } e_C \text{ then } e_T \text{ else } e_E) &= \text{ruleRHSs}(p, e_T) \cup \text{ruleRHSs}(p, e_E) \quad (\text{rules for both sub-exprs.})
\end{align*}
\]

Figure A.10: Defining the right-hand sides of the rewrite rules generated to model a given higher-order expression.
getRules : \(2^P \rightarrow 2^{\text{Rule}}\) returns the set of rewrite rules for a set of grammar productions.

getRules(P) = \{(p, r) \mid p \in P, r \in \text{ruleRHSs}(p, e), (x = e) \in \text{defs}(p)\}

where ruleRHSs (defined in Figure A.10) returns the rules for a higher-order definition in a production. Thus if \(P\) is the set of grammar productions, then getRules(P) is the set of rewrite rules generated for the grammar.

We define two operators \(\Rightarrow\) and \(\rightarrow\) to indicate how terms are derived using these rules. We have \(t \Rightarrow s\) if \(t\) rewrites to \(s\) via a rule in getRules(P).

\[t \Rightarrow s\] if \(t\) rewrites to \(s\) via a rule in getRules(P).

\[t \Rightarrow^* s\] if applying a rule (given its RHS \(r\)) on a tree term \(t\) results in tree term \(s\).

\[t \Rightarrow+ s\] if applying a rule (given its RHS \(r\)) on a tree term \(t\) results in tree term \(s\) at least once.

A.4.2 Deriving Evaluated Local Trees from their Parent Sub-Trees Using the Rules

For each local tree creation step during evaluation, we can derive the evaluated tree value via a non-empty rewrite rule sequence from the local’s parent’s sub-tree, as explained
$eval_\alpha$ 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.

$eval_\alpha : N \rightarrow Expr \rightarrow Attribution \rightarrow Term$

For an expression $e$ on a node $n$ where $\text{type}_e(e) \in (NT \cup T)$, $eval_\alpha(n, e, \Gamma)$ is defined as follows:

- $eval_\alpha(n, \#i, \Gamma) = \text{term}(\text{child}(n, i))$ (i\textsuperscript{th} child tree)
- $eval_\alpha(n, \#i.a, \Gamma) = \text{term}(\text{child}(n, i))$ (i\textsuperscript{th} child tree, computes $a$)
- $eval_\alpha(n, 1.a, \Gamma) = eval_\alpha(n, e_L, \Gamma)$ (from local def., $a$’s tree)
- $eval_\alpha(n, q(e_1, ..., e_n), \Gamma) = q(eval_\alpha(n, e_1, \Gamma), ..., eval_\alpha(n, e_n, \Gamma))$ (rules model tree creation)
- $eval_\alpha(n, c, \Gamma) = c$ (terminal symbol)
- $eval_\alpha(n, \text{if } e_C \text{ then } e_T \text{ else } e_E, \Gamma) = \begin{cases} eval_\alpha(n, e_T, \Gamma) & \text{if } eval(n, e_C, \Gamma) = \text{true} \\ eval_\alpha(n, e_E, \Gamma) & \text{otherwise} \end{cases}$ (conditional expr. is evaluated)

Figure A.11: The first term in an evaluated attribute instance’s rewrite sequence.
in Section 4.4.2. The first term in the rewrite sequence represents the tree value after all conditions in its defining expression have been evaluated. The first term thus is a partially evaluated version of the expression in which all conditions have been evaluated, but in which attribute instance accesses are not computed and are represented by the sub-trees on which they are evaluated. This term is derivable from the tree term of the attribute instance’s defining node, via the one rule that corresponds to the actual values of its conditional expressions.

The first term in the rewrite sequence is defined by the function \( \text{eval}_\alpha \) in Figure A.11 to correspond to the definition in Figure A.10 of each production’s rules.

**Lemma \( \alpha' \):** For any evaluable higher-order attribute instance \( n \# a \) with defining expression \( e \) in a state \( \langle T, \Gamma \rangle \), we can rewrite \( n \)'s sub-tree to \( \text{eval}_\alpha(n, e, \Gamma) \) via at least one rewrite rule in \( \text{getRules}(P) \), i.e.,

\[
\text{term}(n) \Rightarrow \text{eval}_\alpha(n, e, \Gamma)
\]

**Proof:** This is implied by **Lemma \( \alpha \).**

**Lemma \( \alpha \):** For any evaluable higher-order expression \( e \), evaluated on a node \( n \) in a state \( \langle T, \Gamma \rangle \), we can rewrite \( n \)'s sub-tree to \( \text{eval}_\alpha(n, e, \Gamma) \) via at least one rewrite rule in \( \text{ruleRHSs}(\text{prod}(n), e) \).

At any state \( \langle T, \Gamma \rangle \) and higher-order expression \( e \) where
- \( t' \in T \),
- \( n \in \text{nodes}(t') \),
- \( \text{type}_e(\text{prod}(n), e) \in NT \cup T \) and
- \( \forall n' \# a' \in \text{dep}(n, e, \Gamma) \cdot \Gamma[n'[\# a']] \neq \bot \),
we have
- \( \text{term}(n) \xrightarrow{r} \text{eval}_\alpha(n, e, \Gamma) \) for some \( r \in \text{ruleRHSs}(\text{prod}(n), e) \).

**Proof:** Proof is by induction on the structure of \( e \).
Let children(n) = n₁, ..., nₚ and p = prod(n).

Base Cases:

- e is #i: term(n) #i \rightarrow term(n_i) = eval_α(n, #i, Γ)
  where \text{ruleRHSs}(p, #i) = \{#i\}.

- e is c: term(n) c \rightarrow c = eval_α(n, c, Γ)
  where \text{ruleRHSs}(p, c) = \{c\}.

- e is #i.aS: term(n) #i \rightarrow term(n_i) = eval_α(n, #i.aS, Γ)
  where \text{ruleRHSs}(p, #i.aS) = \{#i\}.

Inductive Cases:

- e is l.aS:
  - Assume
    - (l = e_L) ∈ \text{defs}(p) and
    - term(n) \rightarrow_L eval_α(n, e_L, Γ) for some r_L ∈ \text{ruleRHSs}(p, e_L).
  - Since
    - eval_α(n, l.aS, Γ) = eval_α(n, e_L, Γ) and
    - \text{ruleRHSs}(p, l.aS) = \text{ruleRHSs}(p, e_L),
    we have term(n) r \rightarrow eval_α(n, e, Γ) where r ∈ \text{ruleRHSs}(p, e).

- e is if e_C then e_T else e_E:
  - Assume
    - term(n) \rightarrow_T eval_α(n, e_T, Γ) for some r_T ∈ \text{ruleRHSs}(p, e_T) and
    - term(n) \rightarrow_E eval_α(n, e_E, Γ) for some r_E ∈ \text{ruleRHSs}(p, e_E).
  - If eval(n, e_C, Γ) = true then eval_α(n, e, Γ) = eval_α(n, e_T, Γ).
  - If eval(n, e_C, Γ) = false then eval_α(n, e, Γ) = eval_α(n, e_E, Γ).
  - Since \text{ruleRHSs}(p, e) = \text{ruleRHSs}(p, e_T) \cup \text{ruleRHSs}(q, e_E),
    we have term(n) r \rightarrow eval_α(n, e, Γ) for some r ∈ \text{ruleRHSs}(p, e).
- \( e \) is \( q(e_1, \ldots, e_{n_q}) \):

- Assume
  - \( \text{term}(n) \xrightarrow{r_1} \text{eval}_\alpha(n, e_1, \Gamma) \) for some \( r_1 \in \text{ruleRHSs}(p, e_1) \),
  - \( \ldots \)
  - \( \text{term}(n) \xrightarrow{r_{n_q}} \text{eval}_\alpha(n, e_{n_q}, \Gamma) \) for some \( r_{n_q} \in \text{ruleRHSs}(p, e_{n_q}) \).

- So we have \( \text{term}(n) \xrightarrow{q(r_1, \ldots, r_{n_q})} \text{eval}_\alpha(n, e_1, \Gamma), \ldots, \text{eval}_\alpha(n, e_{n_q}, \Gamma) \) where \( r_1 \in \text{ruleRHSs}(p, e_1), \ldots, r_{n_q} \in \text{ruleRHSs}(p, e_{n_q}) \).

- Since \( \text{eval}_\alpha(n, q(e_1, \ldots, e_{n_q}), \Gamma) = q(\text{eval}_\alpha(n, e_1, \Gamma), \ldots, \text{eval}_\alpha(n, e_{n_q}, \Gamma)) \) and \( \text{ruleRHSs}(p, q(e_1, \ldots, e_{n_q})) = \{ q(r_1, \ldots, r_{n_q}) | r_i \in \text{ruleRHSs}(p, e_i), 1 \leq i \leq n_q \} \),
  we have \( \text{term}(n) \xrightarrow{r} \text{eval}_\alpha(n, e, \Gamma) \) for some \( r \in \text{ruleRHSs}(p, e) \).

In the first term in the sequence, 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 of the child or local tree terms, additional rewrites can be performed on these sub-terms to generate the actual evaluated attribute instance value. The fact that these additional rewrites can be performed to generate the correct value can be shown inductively on the structure of the defining expression. The inductive assumption is that all previously evaluated tree values have corresponding valid rewrite sequences from their parent subtrees to the created values.

**Lemma \( \sigma' \):** For any evaluable higher-order attribute instance \( n\#a \) with defining expression \( e \) in a state \( \langle T, \Gamma \rangle \), we can rewrite \( \text{eval}_\alpha(n, e, \Gamma) \) to \( n\#a \)'s value, i.e.,

\[
\text{eval}_\alpha(n, e, \Gamma) \implies^* \Gamma[n\#a]
\]

**Proof:** This is implied by **Lemma \( \sigma \).**

**Lemma \( \sigma \):** For any evaluable higher-order expression \( e \), evaluated on a node \( n \) in a state \( \langle T, \Gamma \rangle \), we can rewrite \( \text{eval}_\alpha(n, e, \Gamma) \) to the evaluated value of \( e \) using the
rules in `getRules(P)`, if the evaluated values of `e`’s required instances are similarly derivable from their nodes’ sub-trees.

At any state `<T, Γ>` and higher-order expression `e` where
- `t' ∈ T`,
- `n ∈ nodes(t')`,
- `type_e(prod(n), e) ∈ NT ∪ T` and
- `∀n' #a' ∈ dep(n, e, Γ). (Γ[n' #a'] ≠ ⊥ and term(n')} =⇒ Γ[n' #a']`,
we have
- `eval_α(n, e, Γ) =⇒* eval(n, e, Γ)

Proof: Proof is by induction on the structure of `e`.

Let `children(n) = n₁, ..., nₚ` and `p = prod(n)`.

Base Cases:

- `e` is `#i`:
  - Since `eval_α(n, #i, Γ) = term(n_i) = eval(n, #i, Γ)`,
    we have `eval_α(n, #i, Γ) =⇒* eval(n, #i, Γ)`.

- `e` is `c`:
  - Since `eval_α(n, c, Γ) = c = eval(n, c, Γ)`,
    we have `eval_α(n, c, Γ) =⇒* eval(n, c, Γ)`.

- `e` is `#i.a_S`:
  - Since `n_i #a_S ∈ dep(n, e, Γ) we have term(n_i) =⇒* Γ[n_i #a_S]`.
  - Thus `eval_α(n, #i.a_S, Γ) = term(n_i) =⇒* Γ[n_i #a_S] = eval(n, #i.a_S, Γ)`.
  - So we have `eval_α(n, #i.a_S, Γ) =⇒* eval(n, #i.a_S, Γ)`.

Inductive Cases:

- `e` is `l.a_S`:
- Assume
  - \(1 = e_L\) \(\in\) \text{defs}(p)\) and
  - \(\text{eval}_\alpha(n, e_L, \Gamma) \Rightarrow^* \text{eval}(n, e_L, \Gamma)\).

- Since \((\Gamma[n\#1])#_s \in \text{dep}(n, e, \Gamma)\) we have \(\text{term}(\Gamma[n\#1]) \Rightarrow^* \Gamma[\Gamma[n\#1])#_s]\).

- Thus \(\text{eval}_\alpha(n, l.a_s, \Gamma) = \text{eval}_\alpha(n, e_L, \Gamma) \Rightarrow^* \text{eval}(n, e_L, \Gamma) = \text{term}(\Gamma[n\#1]) \Rightarrow^* \Gamma[\Gamma[n\#1])#_s] = \text{eval}(n, l.a_s, \Gamma)\).

- So we have \(\text{eval}_\alpha(n, l.a_s, \Gamma) \Rightarrow^* \text{eval}(n, l.a_s, \Gamma)\).

- \(e\) is if \(e_C\) then \(e_T\) else \(e_E\):

  - Assume
    - \(\text{eval}_\alpha(n, e_T, \Gamma) \Rightarrow^* \text{eval}(n, e_T, \Gamma)\) and
    - \(\text{eval}_\alpha(n, e_E, \Gamma) \Rightarrow^* \text{eval}(n, e_E, \Gamma)\).

  - If \(\text{eval}(n, e_C, \Gamma) = \text{true}\), then
    - \(\text{eval}_\alpha(n, e, \Gamma) = \text{eval}_\alpha(n, e_T, \Gamma)\) and
    - \(\text{eval}(n, e, \Gamma) = \text{eval}(n, e_T, \Gamma)\).

  - If \(\text{eval}(n, e_C, \Gamma) = \text{false}\), then
    - \(\text{eval}_\alpha(n, e, \Gamma) = \text{eval}_\alpha(n, e_E, \Gamma)\) and
    - \(\text{eval}(n, e, \Gamma) = \text{eval}(n, e_E, \Gamma)\).

  - In either case, \(\text{eval}_\alpha(n, e, \Gamma) \Rightarrow^* \text{eval}(n, e, \Gamma)\).

- \(e\) is \(q(e_1, \ldots, e_{n_q})\):

  - Assume
    - \(\text{eval}_\alpha(n, e_1, \Gamma) \Rightarrow s_1^1 \Rightarrow s_1^m_1 \Rightarrow \text{eval}(n, e_1, \Gamma)\) where \(m_1 \geq 0\),
    - \(\ldots\)
    - \(\text{eval}_\alpha(n, e_{n_q}, \Gamma) \Rightarrow s_q^{n_q} \Rightarrow s_q^{m_{n_q}} \Rightarrow \text{eval}(n, e_{n_q}, \Gamma)\) where \(m_{n_q} \geq 0\).

  - \(\text{eval}_\alpha(n, q(e_1, \ldots, e_{n_q}), \Gamma) = q(\text{eval}_\alpha(n, e_1, \Gamma), \ldots, \text{eval}_\alpha(n, e_{n_q}, \Gamma)) \Rightarrow q(s_1^1, \ldots, \text{eval}_\alpha(n, e_{n_q}, \Gamma)) \Rightarrow \ldots \Rightarrow q(s_q^{m_1}, \ldots, \text{eval}_\alpha(n, e_{n_q}, \Gamma)) \Rightarrow \ldots \Rightarrow q(\text{eval}(n, e_1, \Gamma), \ldots, s_q^{n_q}) \Rightarrow \ldots \Rightarrow q(\text{eval}(n, e_1, \Gamma), \ldots, s_{m_{n_q}}) \Rightarrow q(\text{eval}(n, e_{n_q}, \Gamma)) = \text{eval}(n, q(e_1, \ldots, e_{n_q}), \Gamma).

  - Therefore \(\text{eval}_\alpha(n, e, \Gamma) \Rightarrow^* \text{eval}(n, e, \Gamma)\).
Using Lemma $\alpha'$ and Lemma $\sigma'$, we can derive every higher-order evaluable attribute instance’s value from its defining node’s tree term via a non-empty rewrite sequence. Such instances are either local attributes, or higher-order synthesized instances. The fact that local values are derivable means for every tree creation sequence, there is a rewrite sequence of at least the same length, as shown in the next section.

**Lemma $\omega$:** For any evaluable higher-order attribute instance $n#a$ in a state $\langle T, \Gamma \rangle$, we can derive the evaluated value of $n#a$ from $n$’s sub-tree via a non-empty rewrite sequence.

At any state $\langle T, \Gamma \rangle$ where
- $t' \in T$,
- $n#a \in \text{instances}(t')$,
- $\forall (n'#a') \in \text{dep}(n, e, \Gamma). \Gamma[n'#a'] \neq \bot$ and
- either
  - $a \in A_S, \text{type}_a(a) \in (NT \cup T), (#0.a = e) \in \text{defs(prod}(n)))$ or
  - $a \in L, (a = e) \in \text{defs(prod}(n))$,
we have
- $\text{term}(n) \Rightarrow^+ \Gamma[n#a]$.

**Proof:** Proof is by induction on the number of higher-order instances evaluated.

- By Lemma $\alpha$, $\text{term}(n) \xrightarrow{r} \text{eval}_\alpha(n, e, \Gamma)$ for some $r \in \text{ruleRHSs}(p, e)$.
- Since $\forall r \in \text{ruleRHSs}(p, e) . (p,r) \in \text{getRules}(P)$, we have $\text{term}(n) \Rightarrow \text{eval}_\alpha(n, e, \Gamma)$.

**Base Case:**

- For the first higher-order instance evaluated, we have $\text{dep}(n, e, \Gamma) = \{ \}$.  
- We therefore trivially have
  $$\forall n'#a' \in \text{dep}(n, e, \Gamma) . (\Gamma[n'#a'] \neq \bot \text{ and } \text{term}(n') \Rightarrow^+ \Gamma[n'#a']).$$
- By Lemma $\sigma$, $\text{eval}_\alpha(n, e, \Gamma) \Rightarrow^* \text{eval}(n, e, \Gamma)$.
- Therefore $\text{term}(n) \Rightarrow^+ \text{eval}(n, e, \Gamma)$. 

Inductive Case:

- For an evaluable higher-order instance, all required instances are higher-order.
- Therefore by the inductive assumption,
  \[ \forall (n', \#a') \in \text{dep}(n, e, \Gamma). (\Gamma[n', \#a'] \neq \bot \text{ and } \text{term}(n') \rightarrow^+ \Gamma[n', \#a']) \].
- By Lemma \(\sigma\), \(\text{eval}_\alpha(n, e, \Gamma) \rightarrow^* \text{eval}(n, e, \Gamma)\).
- Therefore \(\text{term}(n) \rightarrow^+ \text{eval}(n, e, \Gamma)\).

A.4.3 Constructing a Rewriting Sequence for a Tree Creation Sequence

Given a tree creation sequence (corresponding to a path in the tree of locals), we can construct a rewrite sequence of at least the same length. As each tree is evaluated on a node of its predecessor, the first term of the rewrite sequence that models this tree creation step (given by Lemma \(\omega\)) is a sub-term of the predecessor tree. Thus the predecessor tree itself can be rewritten to a term in which the parent sub-tree is replaced by the local. This new term contains the new local as a sub-term, and therefore the process can be continued for the next evaluated local. As the rewrite sequence corresponding to each local’s evaluation is non-empty, we can construct a rewrite sequence to model the entire tree creation sequence that is at least as long as the tree creation sequence.

**Lemma \(\rho\):** For a tree creation sequence
\[ \langle t_0, \Gamma_0 \rangle, \langle t_1, \Gamma_1, n_1 \# l_1 \rangle, \langle t_2, \Gamma_2, n_2 \# l_2 \rangle, \ldots \]
we can define a function \(R : N \rightarrow \text{Term}\) so that for \(i > 0\), \(R(t_{i-1}) \rightarrow^+ R(t_i)\), i.e., there is a non-empty rewrite sequence from \(R(t_{i-1})\) to \(R(t_i)\) of rules in \(\text{getRules}(P)\).

**Proof:** Proof is by construction.

We can define an \(R : N \rightarrow \text{Term}\) such that for \(i \geq 0\)
- \(\text{term}(t_i)\) is a sub-term of \(R(t_i)\).
- \(R(t_{i-1}) \rightarrow^+ R(t_i)\) if \(i > 0\).
This would imply Lemma $\rho$.

We define $R$ inductively such that these conditions hold as follows:

**Base Case:** $i = 0$
- Let $R(t_0)$ be $\text{term}(t_0)$.
- $\text{term}(t_0) = R(t_0)$ is trivially a sub-term of $R(t_0)$.

**Inductive Case:** $i > 0$
- By the inductive assumption, $\text{term}(t_i)$ is a sub-term of $R(t_i)$ and $R(t_{i-1}) \Rightarrow^+ R(t_i)$.
- We need to define $R(t_{i+1})$ such that $\text{term}(t_{i+1})$ is a sub-term of $R(t_{i+1})$ and $R(t_i) \Rightarrow^+ R(t_{i+1})$.
- Since $n_{i+1} \in \text{nodes}(t_i)$, $\text{term}(n_{i+1})$ is a sub-term of $\text{term}(t_i)$ (and hence of $R(t_i)$).
- Since $n_{i+1} \# l_{i+1}$ is an evaluable higher-order attribute instance, we have $\text{term}(n_{i+1}) \Rightarrow^+ \text{term}(t_{i+1})$ by Lemma $\omega$.
- Let $R(t_{i+1})$ be $R(t_i)[\text{term}(n_{i+1}) \leadsto \text{term}(t_{i+1})]$.
- Then $\text{term}(t_{i+1})$ is a sub-term of $R(t_{i+1})$ and $R(t_i) \Rightarrow^+ R(t_{i+1})$.

Thus if the rules terminate, then tree creation, and therefore attribute evaluation cannot continue indefinitely.

**Theorem II:** If the rewrite rules in $\text{getRules}(P)$ are terminating, there is no infinite tree creation sequence, and hence no improper evaluation sequence.

**Proof:** Proof is by contradiction.
- Assume there is an improper evaluation sequence.
- By Theorem I, there is an infinite tree creation sequence $\langle t_0, \Gamma_0 \rangle, \langle t_1, \Gamma_1, n_1 \# l_1 \rangle, \langle t_2, \Gamma_2, n_2 \# l_2 \rangle, \ldots$
- By Lemma $\rho$ therefore, we can define a function $R$ so that for $i > 0$, $R(t_{i-1}) \Rightarrow^+ R(t_i)$. 
• Thus there is an infinite rewrite sequence $R(t_0) \Rightarrow^+ R(t_1) \Rightarrow^+ R(t_2) \Rightarrow^+ \ldots$ of rules in $\text{getRules}(P)$.

• Thus if there is an improper evaluation sequence, the rules are non-terminating.

• Thus if the rules are terminating, there is no improper evaluation sequence.

A.5 Section 4.5 (Ordering Trees to Limit Accesses to Inherited Attributes)

A.5.1 Ordering Non-Terminals to Restrict Inherited Access During Tree Creation

| RuleRHS ::= INH. |
| ruleRHSs(p, #0.a_I) = \{INH\}. |
| Term ::= INH. |
| instances(INH) = \{ \}. |
| $t^{\text{INH}} \leftarrow\text{INH.} |
| \text{eval}_\alpha(n, #0.a_I, \Gamma) = \text{INH.} |

Figure A.12: Extending the definitions of rewrite rules and terms to handle grammars with higher-order inherited attributes.

As explained in Section 4.5.1, the term rewriting rules described above cannot model tree creation in the presence of higher-order inherited attributes. We first extend our definitions of rewrite terms and rules to handle grammars with higher-order inherited accesses. The updated definitions are shown in Figure A.12. The right-hand sides of rewrite rules now include the place-holder term INH, which represents higher-order inherited attributes. This term is needed since the rule generator does not have access to the context in which the inherited attribute is defined. Also shown is the updated version of the function ruleRHSs (defined in Figure A.10) that generates the rules for model a given higher-order expression. We also extend Term to include INH so that the results of applying rewrites to tree terms are still tree terms. Finally, we extend the definition of eval_\alpha given in Figure A.11. With these extended definitions, we can
generate rules for grammars with higher-order inherited attributes. And as described in Section 4.5.1, the rules cannot model tree creation steps with higher-order inherited attributes as none of the rules has \texttt{INH} on the left-hand side.

### A.5.2 Ordering Non-Terminals to Restrict Inherited Access During Tree Creation

We present an analysis that checks the grammar for restrictions that ensure that the number of inherited accesses in a tree creation sequence is bounded. The restrictions ensure that there is a function \( \mathcal{L} \) that maps the grammar’s non-terminals to a finite total order \( \textit{leq} \) such that the following conditions are satisfied:

\textbf{Defn.} \( \Lambda \):

1. \( \mathcal{L}(X) \leq \mathcal{L}(Y) \) if \( \exists p \in P . X = \text{lhs}(p), Y \in \text{rhs}(p) \)
2. \( \mathcal{L}(X) \leq \mathcal{L}(Y) \) if \( \exists a_S \in A_S . a_S@X, \text{type}_a(a_S) = Y \)
3. \( \mathcal{L}(X) \leq \mathcal{L}(Y) \) if \( \exists p \in P, 1 \in \text{locals}(p) . X = \text{lhs}(p), \text{type}_1(1) = Y \)
4. \( \mathcal{L}(X) < \mathcal{L}(Y) \) if \( \exists a_I \in A_I . a_I@X, \text{type}_a(a_I) = Y \)

An algorithm for \( \mathcal{L} \) is given in Section 4.5.4. It assigns non-negative sizes to the non-terminals based on how higher-order values are created in the grammar. If the grammar non-terminals can be ordered and satisfy the properties above, we can order the trees in a tree creation sequence, by comparing their root non-terminals.

\textbf{Lemma \( \lambda \)}: Every tree created as a local is at least as large as its predecessor tree. Further, if it is of the same size, then the local is not set to the value of an inherited attribute.

\textbf{Proof}:

- Assume a function \( \mathcal{L} \) from \( NT \) to a finite total order that satisfies the conditions in Defn. \( \Lambda \).
- Consider a tree creation step \( \langle t_{i-1}, \Gamma_{i-1} \rangle, \langle t_i, \Gamma_i \rangle \) where
- $t_i = \text{newTree}(\text{eval}(n_i, e_i, \Gamma'))$,
- $n_i \# l_i \in \text{instances}(t_{i-1})$,
- $(l_i = e_i) \in \text{defs}(\text{prod}(n_i))$, and
- $\Gamma'$ is $\Gamma_i$'s predecessor evaluation state, which is not the same as $\Gamma_{i-1}$ if there are intervening non-local evaluating steps.

- By Defn. A(1), non-terminal symbols are non-decreasing from the root to the leaves of the parent tree, i.e., $\mathcal{L}(\text{symbol}(t_{i-1})) \leq \mathcal{L}(\text{symbol}(n_i))$.

- By Defn. A(2) and Defn. A(3), the new tree's root symbol is at least as large as the symbol of the node on which it was created, i.e., $\mathcal{L}(\text{symbol}(n_i)) \leq \mathcal{L}(\text{symbol}(t_i))$.

- Thus the root symbol of the new tree is at least as large as the tree on which it was evaluated, i.e., $\mathcal{L}(\text{symbol}(t_{i-1})) \leq \mathcal{L}(\text{symbol}(t_i))$.

- If $e_i$ is $\#0.a_I$ then $\mathcal{L}(\text{symbol}(t_i)) > \mathcal{L}(\text{symbol}(t_{i-1}))$ by Defn. A(4).

- Thus if $\mathcal{L}(\text{symbol}(t_i)) = \mathcal{L}(\text{symbol}(t_{i-1}))$ then $e_i$ is not $\#0.a_I$.

A.5.3 For an Infinite Tree Creation Sequence, There is an Infinite Constant Tree Creation Sequence

**Lemma $\kappa$:** If all non-terminals can be assigned valid inheritance levels, then for any infinite tree creation sequence, there exists an infinite constant tree creation sequence.

**Proof:** Proof is by construction.

- Assume a function $\mathcal{L}$ from $NT$ to a finite total order that satisfies the conditions in Defn. A.
- Assume an infinite tree creation sequence
  \[
  \langle t_0, \Gamma_0 \rangle, \langle t_1, \Gamma_1, n_1 \# l_1 \rangle, \langle t_2, \Gamma_2, n_2 \# l_2 \rangle, \ldots \text{ where for } i > 0,
  \]
  \[
  - t_i = \text{newTree}(\text{eval}(n_i, e_i, \Gamma')),
  \]
\[ n_i \mathbb{1}_i \in \text{instances}(t_{i-1}), \]
\[ (1_i = e_i) \in \text{defs}(\text{prod}(n_i)), \]
\[ \Gamma' \text{ is } \Gamma_i \text{'s predecessor evaluation state, which is not the same as } \Gamma_{i-1} \text{ if there are intervening non-local steps.} \]

- By Lemma \( \lambda \), the trees are non-decreasing, i.e., \( L(\text{symbol}(t_i)) \leq L(\text{symbol}(t_{i+1})) \) for \( i \geq 0 \).
- Since the total order is finite, there is a non-terminal with no larger non-terminal.
- Thus there is a tree in the sequence beyond which the root non-terminals are of the same size.
- Thus for an infinite tree creation sequence, there is an infinite constant tree creation sequence, i.e., \( \exists k \) such that
  \[ \langle t_k, \Gamma_k \rangle, \langle t_{k+1}, \Gamma_{k+1}, n_{k+1} \mathbb{1}_{k+1} \rangle, \langle t_{k+2}, \Gamma_{k+2}, n_{k+2} \mathbb{1}_{k+2} \rangle, \ldots \text{ is a tree creation sequence and for } j > k, \]
  - \( t_j = \text{newTree}(\text{eval}(n_j, e_j, \Gamma')) \),
  - \( n_j \mathbb{1}_j \in \text{instances}(t_{j-1}) \),
  - \( (1_j = e_j) \in \text{defs}(\text{prod}(n_j)) \),
  - \( \Gamma' \text{ is } \Gamma_j \text{'s predecessor evaluation state, and} \)
  - \( L(\text{symbol}(t_j)) = L(\text{symbol}(t_{j-1})) \).

\section*{A.5.4 The Rules Do Model Constant Tree Creation Steps}

We can show that the rules do model the tree creation steps in a constant tree creation sequence. We assume that the roots of these trees are of the same size as the non-terminal \( K \). Since the rewrite terms represent inherited accesses with the placeholder \text{INH}, we can no longer generate the actual local trees using the rules. Rather we generate “pruned” versions of the trees, in which some sub-trees are replaced by \text{INH}. We introduce the \( \sqsubseteq_K \) symbol to represent the relation between a tree and its pruned versions.

\[ t_1 \sqsubseteq_K t_2 \text{ if } t_1 \text{ is a pruned version of } t_2 \text{ in which some sub-terms with root symbols greater than } K \text{ are replaced with } \text{INH}. \]

\( \sqsubseteq_K \subseteq \text{Term } \times \text{Term} \)
• \( c \sqsubseteq_K c \)
• \( s_1 \sqsubseteq_K t_1, \ldots, s_nq \sqsubseteq_K t_nq \)
  \( q(s_1, \ldots, s_nq) \sqsubseteq_K q(t_1, \ldots, t_nq) \)
• \( L(\text{lhs}(q)) > L(K) \)
• \( \text{INH} \sqsubseteq_K q(t_1, \ldots, t_nq) \)

The root symbols of the \text{INH} sub-trees are greater than \( K \). In effect, we ignore those nodes in the trees whose symbols are greater than \( K \). All nodes in the original trees that are the same size as \( K \) are present in the pruned trees. If \( s \sqsubseteq_K q(t_1, \ldots, t_nq) \) where \( L(\text{lhs}(q)) \leq L(K) \), then \( s = q(s_1, \ldots, s_nq) \) where \( s_1 \sqsubseteq_K t_1, \ldots, s_nq \sqsubseteq_K t_nq \). Rewrite rules can therefore still be applied to such nodes.

Thus the rules can generate a pruned version of a local tree from any pruned version of its parent sub-tree. As in the case without higher-order inherited attributes, the rewrite sequence for each local is constructed in stages. For a particular constant tree-creation step, the first term of the sequence is a pruned partially evaluated term in which conditions are evaluated. Further, if the local is of the same size as \( K \), then this first term will not be \text{INH} and can be rewritten. The rest of the sequence rewrites the pruned sub-trees of this first term, on which synthesized instances are defined, to pruned versions of their evaluated values. We can thus generate the rewrite sequence for any higher-order evaluation step. We define corresponding versions of \textbf{Lemma} \( \alpha' \), \textbf{Lemma} \( \alpha \), \textbf{Lemma} \( \sigma' \), \textbf{Lemma} \( \sigma \) and \textbf{Lemma} \( \omega \) as follows.

\textbf{Lemma} \( \alpha'^{\text{INH}} \): For any evaluable higher-order attribute instance \( n\#a \) with defining expression \( e \) in a state \( \langle \mathcal{T}, \Gamma \rangle \) where \text{symbol}(n) = K, we can rewrite any \( u \) where \( u \sqsubseteq_K \text{term}(n) \) to \( v \) where \( v \sqsubseteq_K \text{eval}_a(n, e, \Gamma) \) via at least one rewrite rule in \texttt{getRules}(P). Further if \( a \)'s type is of the same size as \( K \), then \( v \) is not \text{INH}, i.e.,

if \( u \sqsubseteq_K \text{term}(n) \) then \( u \Rightarrow v \) where \( v \sqsubseteq_K \text{eval}_a(n, e, \Gamma) \) and

\( v \neq \text{INH} \) if \( a \)'s non-terminal is of the same size as \( K \).

\textbf{Proof}: This is implied by \textbf{Lemma} \( \alpha'^{\text{INH}} \).
**Lemma** \( \alpha^{INH} \): For any evaluable higher-order expression \( e \), evaluated on a node \( n \) in a state \( \langle T, \Gamma \rangle \) where \( \text{symbol}(n) = K \), we can rewrite any \( u \) where \( u \sqsubseteq_K \text{term}(n) \) to \( v \) where \( v \sqsubseteq_K \text{eval}_\alpha(n, e, \Gamma) \) via at least one rewrite rule in \( \text{ruleRHSs}(\text{prod}(n), e) \). Further if \( e \)'s type is of the same size as \( K \), then \( v \) is not \( \text{INH} \).

At any state \( \langle T, \Gamma \rangle \) and higher-order expression \( e \) where
- \( t' \in T \),
- \( n \in \text{nodes}(t') \),
- \( \text{type}_e(\text{prod}(n), e) \in NT \cup T \),
- \( \forall n' \#a' \in \text{dep}(n, e, \Gamma) : \Gamma[n' \#a'] \neq \bot \),
- \( \text{symbol}(n) = K \) and
- \( u \sqsubseteq_K \text{term}(n) \)
we have
- \( u \xrightarrow{r} v \) where \( v \sqsubseteq_K \text{eval}_\alpha(n, e, \Gamma) \) for some \( r \in \text{ruleRHSs}(\text{prod}(n), e) \), and
- if \( \mathcal{L}(\text{type}_e(e)) = \mathcal{L}(K) \) then \( v \neq \text{INH} \).

**Proof:** Proof is by induction on the structure of \( e \).

Let \( \text{children}(n) = n_1, \ldots, n_{n_p} \) and \( p = \text{prod}(n) \).

We have \( u = p(u_1, \ldots, u_{n_p}) \) where \( u_1 \sqsubseteq_K \text{term}(n_1), \ldots, u_{n_p} \sqsubseteq_K \text{term}(n_{n_p}) \).

**Base Cases:**

- \( e \) is \( \#i \):
  - \( u \not\leadsto u_i \) where \( u_i \sqsubseteq_K \text{term}(n_i) = \text{eval}_\alpha(n, \#i, \Gamma) \) and \( \text{ruleRHSs}(p, \#i) = \{\#i\} \).
  - If \( \mathcal{L}(\text{symbol}(n_i)) = \mathcal{L}(K) \) then \( u_i \neq \text{INH} \).

- \( e \) is \( c \):
  - \( u \xrightarrow{c} c \) where \( c \sqsubseteq_K c = \text{eval}_\alpha(n, c, \Gamma) \)
and \text{ruleRHSs}(p, c) = \{c\}.
\begin{itemize}
\item \(c \neq \text{INH}\).
\end{itemize}
- \(e\) is \#i.a_s:
\begin{itemize}
\item \(u \xrightarrow{\#i} u_i\) where \(u_i \sqsubseteq_K \text{term}(n_i) = \text{eval}_\alpha(n, \#i.a_s, \Gamma)\) and \(\text{ruleRHSs}(p, \#i.a_s) = \{\#i\}\).
\item If \(\mathcal{L}(\text{type}_a(a_s)) = \mathcal{L}(K)\) then \(\mathcal{L}(\text{symbol}(n_i)) = \mathcal{L}(K)\) and \(u_i \neq \text{INH}\).
\end{itemize}
- \(e\) is \#0.a_t:
\begin{itemize}
\item \(\mathcal{L}(\text{type}_a(a_t)) > \mathcal{L}(K)\).
\item \(u \xrightarrow{\text{INH}} \text{INH}\) where \(\text{INH} \sqsubseteq_K \text{term}(n_i) = \text{eval}_\alpha(n, \#i.a_s, \Gamma)\) and \(\text{ruleRHSs}(p, \#0.a_t) = \{\text{INH}\}\).
\end{itemize}

\textbf{Inductive Cases:}
- \(e\) is 1.a_s:
\begin{itemize}
\item Assume
- \((1 = e_L) \in \text{defs}(p)\) and
- \(u \xrightarrow{r_L} v_L\) for some \(v_L \sqsubseteq_K \text{eval}_\alpha(n, e_L, \Gamma)\) and \(r_L \in \text{ruleRHSs}(p, e_L)\).
\item Since
- \(\text{eval}_\alpha(n, 1.a_s, \Gamma) = \text{eval}_\alpha(n, e_L, \Gamma)\) and
- \(\text{ruleRHSs}(p, 1.a_s) = \text{ruleRHSs}(p, e_L)\),
we have \(u \xrightarrow{r} v_L\) for some \(v_L \sqsubseteq_K \text{eval}_\alpha(n, e, \Gamma)\) and \(r \in \text{ruleRHSs}(p, e)\).
\item If \(\mathcal{L}(\text{type}_a(a_s)) = \mathcal{L}(K)\) then \(\mathcal{L}(\text{type}_a(e_L)) = \mathcal{L}(K)\) and \(v_L \neq \text{INH}\).
\end{itemize}
- \(e\) is \(\text{if } e_C \text{ then } e_T \text{ else } e_E\):
\begin{itemize}
\item Assume
- \(u \xrightarrow{r_T} v_T\) for some \(v_T \sqsubseteq_K \text{eval}_\alpha(n, e_T, \Gamma)\) and \(r_T \in \text{ruleRHSs}(p, e_T)\) and
- \(u \xrightarrow{r_E} v_E\) for some \(v_E \sqsubseteq_K \text{eval}_\alpha(n, e_E, \Gamma)\) and \(r_E \in \text{ruleRHSs}(p, e_E)\).
\item If \(\text{eval}(n, e_C, \Gamma) = \text{true}\) then \(\text{eval}_\alpha(n, e, \Gamma) = \text{eval}_\alpha(n, e_T, \Gamma)\).
\item If \(\text{eval}(n, e_C, \Gamma) = \text{false}\) then \(\text{eval}_\alpha(n, e, \Gamma) = \text{eval}_\alpha(n, e_E, \Gamma)\).
\item Since \(\text{ruleRHSs}(p, e) = \text{ruleRHSs}(p, e_T) \cup \text{ruleRHSs}(p, e_E)\),
we have \(u \xrightarrow{r} v\) for some \(v \sqsubseteq_K \text{eval}_\alpha(n, e, \Gamma)\) and \(r \in \text{ruleRHSs}(p, e)\).
• If $\mathcal{L}(\text{type}_e(e_T)) = \mathcal{L}(\text{type}_e(e_E)) = \mathcal{L}(K)$ then $v_T \neq \text{INH}$ and $v_E \neq \text{INH}$ and so $v \neq \text{INH}$.

- $e$ is $q(e_1, ..., e_{n_q})$

- Assume

  - $u \xrightarrow{r_1} v_1$ for some $v_1 \sqsubseteq_K \text{eval}_\alpha(n, e_1, \Gamma)$ and $r_1 \in \text{ruleRHSs}(p, e_1)$.
  
  - ...

  - $u \xrightarrow{r_{n_q}} v_{n_q}$ for some $v_{n_q} \sqsubseteq_K \text{eval}_\alpha(n, e_{n_q}, \Gamma)$ and $r_{n_q} \in \text{ruleRHSs}(p, e_{n_q})$.

- Let $v$ be $q(v_1, ..., v_{n_q})$.

- $v \sqsubseteq_K q(\text{eval}_\alpha(n, e_1, \Gamma), ..., \text{eval}_\alpha(n, e_{n_q}, \Gamma))$.

- So we have $u \xrightarrow{q(r_1, ..., r_{n_q})} v$ for some $v \sqsubseteq_K q(\text{eval}_\alpha(n, e_1, \Gamma), ..., \text{eval}_\alpha(n, e_{n_q}, \Gamma))$ and $r_1 \in \text{ruleRHSs}(p, e_1), ..., r_{n_q} \in \text{ruleRHSs}(p, e_{n_q})$.

- Since $\text{eval}_\alpha(n, q(e_1, ..., e_{n_q}), \Gamma) = q(\text{eval}_\alpha(n, e_1, \Gamma), ..., \text{eval}_\alpha(n, e_{n_q}, \Gamma))$ and $\text{ruleRHSs}(p, q(e_1, ..., e_{n_q})) = \{ q(r_1, ..., r_{n_q}) \mid r_i \in \text{ruleRHSs}(p, e_i), 1 \leq i \leq n_q \}$, we have $u \xrightarrow{r} v$ for some $v \sqsubseteq_K \text{eval}_\alpha(n, e, \Gamma)$ and $r \in \text{ruleRHSs}(p, e)$.

- $v \neq \text{INH}$.

Lemma $\sigma^{I\text{NH}}$: For any evaluable higher-order attribute instance $n\#a$ with defining expression $e$ in a state $(\mathcal{T}, \Gamma)$ where symbol$(n) = K$, we can rewrite any $u$ where $u \sqsubseteq_K \text{eval}_\alpha(n, e, \Gamma)$ to $v$ where $v \sqsubseteq_K \Gamma[n\#a]$, using the rules in getRules($P$). Further if $a$'s type is of the same size as $K$, then $v$ is not $\text{INH}$, i.e.,

if $u \sqsubseteq_K \text{eval}_\alpha(n, e, \Gamma)$ then $u \Rightarrow^* v$ where $v \sqsubseteq_K \Gamma[n\#a]$ and

$(v \neq \text{INH}$ if $a$'s non-terminal is of the same size as $K$).

Proof: This is implied by Lemma $\sigma^{I\text{NH}}$.

Lemma $\sigma^{I\text{NH}}$: For any evaluable higher-order expression $e$, evaluated on a node $n$ in a state $(\mathcal{T}, \Gamma)$, we can rewrite any $u$ where $u \sqsubseteq_K \text{eval}_\alpha(n, e, \Gamma)$ to $v$ where $v \sqsubseteq_K \text{eval}(n, e, \Gamma)$, using the rules in getRules($P$), if the evaluated values of $e$'s
required instances are similarly derivable from their nodes’ sub-trees. Further if e’s
type is of the same size as K, then v is not INH.

At any state \( \langle T, \Gamma \rangle \) and higher-order expression e where
- \( t' \in T, n \in \text{nodes}(t') \), \( \text{type}_e(\text{prod}(n), e) \in NT \cup T \),
- \( \mathcal{L}($\text{symbol}(n)$) = \mathcal{L}($\text{type}_e(\text{prod}(n), e)$) = \mathcal{L}(K) \) for some \( K \in NT \),
- \( u \sqsubseteq_K \text{eval}_\alpha(n, e, \Gamma) \) and
  - \( \forall n' \#a' \in \text{dep}(n, e, \Gamma) \). (\( \Gamma[n' \#a'] \neq \bot \) and
    - for any \( u' \sqsubseteq_K \text{term}(n') \) we have \( u' \implies^+ v' \) for some \( v' \sqsubseteq_K \Gamma[n' \#a'] \),
    and if \( \mathcal{L}(\text{type}_e(a)) = \mathcal{L}(K) \) then \( v' \neq \text{INH} \).
we have
- \( u \implies^* v \) where \( v \sqsubseteq_K \text{eval}(n, e, \Gamma) \).
- if \( \mathcal{L}(\text{type}_e(e)) = \mathcal{L}(K) \) then \( v \neq \text{INH} \).

\textbf{Proof:} Proof is by induction on the structure of e.

Let \( \text{children}(n) = n_1, ..., n_{n_p} \) and \( p = \text{prod}(n) \).

\textbf{Base Cases:}

- e is \#i:
  - Since \( u \sqsubseteq_K \text{eval}_\alpha(n, \#i, \Gamma) = \text{term}(n_i) = \text{eval}(n, \#i, \Gamma) \),
    we have \( u \implies^* v \) where \( v \sqsubseteq_K \text{eval}(n, \#i, \Gamma) \).
  - If \( \mathcal{L}(\text{type}_e(e)) = \mathcal{L}(K) \) then \( u \neq \text{INH} \) and \( v \neq \text{INH} \).
- e is c:
  - Since \( u \sqsubseteq_K \text{eval}_\alpha(n, c, \Gamma) = c = \text{eval}(n, c, \Gamma) \),
    we have \( u \implies^* v \) where \( v \sqsubseteq_K \text{eval}(n, c, \Gamma) \).
  - \( c \neq \text{INH} \).
- e is \#i.a$\,$: 
• If \( u = \text{INH} \)
  - \( \text{INH} \sqsubseteq_K \text{eval}_\alpha(n, \#i.a_S, \Gamma) \) and so \( \text{INH} \sqsubseteq_K \text{eval}(n, \#i.a_S, \Gamma) \).
  - So we have \( u \Rightarrow^* v \) for some \( v \sqsubseteq_K \text{eval}(n, \#i.a_S, \Gamma) \).
  - In the rest of the proof, we assume \( u \neq \text{INH} \).
• Since \( n_i \#a_S \in \text{dep}(n, e, \Gamma) \), for any \( u_i \sqsubseteq_K \text{term}(n_i) \) we have
  - \( u_i \Rightarrow^+ v_i \) for some \( v_i \sqsubseteq_K \Gamma[n_i \#a_S] \), and
  - if \( \mathcal{L}(\text{type}_S(a_S)) = \mathcal{L}(K) \) then \( v_i \neq \text{INH} \).
• \( \text{eval}_\alpha(n, \#i.a_S, \Gamma) = \text{term}(n_i) \).
• \( \text{eval}(n, \#i.a_S, \Gamma) = \Gamma[n_i \#a_S] \).
• So we have \( u \Rightarrow^* v \) for some \( v \sqsubseteq_K \text{eval}(n, \#i.a_S, \Gamma) \).
• If \( \mathcal{L}(\text{type}_S(a_S)) = \mathcal{L}(K) \) we have \( v \neq \text{INH} \).

- \( e \) is \( \#0.a_I \):

  • \( \text{INH} \sqsubseteq_K \text{eval}_\alpha(n, \#0.a_I, \Gamma) \) and so \( \text{INH} \sqsubseteq_K \text{eval}(n, \#0.a_I, \Gamma) \).
  • So we have \( u \Rightarrow^* v \) for some \( v \sqsubseteq_K \text{eval}(n, \#0.a_I, \Gamma) \).

Inductive Cases:

- \( e \) is \( 1.a_S \):

  • If \( u = \text{INH} \)
  - \( \text{INH} \sqsubseteq_K \text{eval}_\alpha(n, 1.a_S, \Gamma) \) and so \( \text{INH} \sqsubseteq_K \text{eval}(n, 1.a_S, \Gamma) \).
  - So we have \( u \Rightarrow^* v \) for some \( v \sqsubseteq_K \text{eval}(n, 1.a_S, \Gamma) \).
  - In the rest of the proof, we assume \( u \neq \text{INH} \).
  • Assume
    - \( (1 = e_L) \in \text{defs}(p) \).
    - for any \( u_L \sqsubseteq_K \text{eval}_\alpha(n, e_L, \Gamma) \) we have
      \( u_L \Rightarrow^* v_L \) for some \( v_L \sqsubseteq_K \text{eval}(n, e_L, \Gamma) \), and
    - if \( \mathcal{L}(\text{type}_I(1)) = \mathcal{L}(K) \) then \( v_L \neq \text{INH} \).
  • Since \( (\Gamma[n\#1])\#a_S \in \text{dep}(n, e, \Gamma) \), for any \( u_L \sqsubseteq_K \text{term}(\Gamma[n\#1]) \) we have
    - \( u_L \Rightarrow^+ v \) for some \( v \sqsubseteq_K \Gamma[(\Gamma[n\#1])\#a_S] \), and
    - if \( \mathcal{L}(\text{type}_S(a_S)) = \mathcal{L}(K) \) then \( v \neq \text{INH} \).
  • \( \text{eval}_\alpha(n, 1.a_S, \Gamma) = \text{eval}_\alpha(n, e_L, \Gamma) \).
• \( \text{eval}(n, e_L, \Gamma) = \text{term}([\Gamma[n\#1]]) \).
• \( \text{eval}(n, 1.a_S, \Gamma) = \Gamma[[\Gamma[n\#1]]\#a_S] \).
• So we have \( u \implies^* v \) for some \( v \sqsubseteq_K \text{eval}(n, 1.a_S, \Gamma) \).
• If \( \mathcal{L}((\text{type}_1(a_S)) = \mathcal{L}(K) \), then \( v \neq \text{INH} \).

- \( e \) is \( e_C \) then \( e_T \) else \( e_E \):
  • If \( u = \text{INH} \)
    - \( \text{INH} \sqsubseteq_K \text{eval}_a(n, \text{if } e_C \text{ then } e_T \text{ else } e_E, \Gamma) \) and so
      \( \text{INH} \sqsubseteq_K \text{eval}(n, \text{if } e_C \text{ then } e_T \text{ else } e_E, \Gamma) \).
    - So we have \( u \implies^* v \) for some \( v \sqsubseteq_K \text{eval}(n, \text{if } e_C \text{ then } e_T \text{ else } e_E, \Gamma) \).
    - In the rest of the proof, we assume \( u \neq \text{INH} \).
  • We have the following inductive assumption:
    - for any \( u_T \sqsubseteq_K \text{eval}_a(n, e_T, \Gamma) \) we have \( u_T \implies^* v_T \) for some \( v_T \sqsubseteq_K \text{eval}(n, e_T, \Gamma) \)
    - for any \( u_E \sqsubseteq_K \text{eval}_a(n, e_E, \Gamma) \) we have \( u_E \implies^* v_E \) for some \( v_E \sqsubseteq_K \text{eval}(n, e_E, \Gamma) \)
  • If \( \text{eval}(n, e_C, \Gamma) = \text{true} \), then
    - \( \text{eval}_a(n, e, \Gamma) = \text{eval}_a(n, e_T, \Gamma) \) and
    - \( \text{eval}(n, e, \Gamma) = \text{eval}(n, e_T, \Gamma) \).
  • If \( \text{eval}(n, e_C, \Gamma) = \text{false} \), then
    - \( \text{eval}_a(n, e, \Gamma) = \text{eval}_a(n, e_E, \Gamma) \) and
    - \( \text{eval}(n, e, \Gamma) = \text{eval}(n, e_E, \Gamma) \).
  • In either case, \( u \implies^* v \) for some \( v \sqsubseteq_K \text{eval}(n, e, \Gamma) \).
  • If \( \mathcal{L}(\text{symbol}(n)) = \mathcal{L}(\text{type}_a(e_T)) = \mathcal{L}(\text{type}_a(e_E)) = \mathcal{L}(K) \)
    - \( v_T \neq \text{INH} \) and \( v_E \neq \text{INH} \) and so \( v \neq \text{INH} \).

- \( e \) 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 \( u_i \sqsubseteq_K \text{eval}_a(n, e_i, \Gamma) \) we have \( u_i \implies^* v_i \) for some \( v_i \sqsubseteq_K \text{eval}(n, e_i, \Gamma) \)
  • Let \( v \) be \( q(v_1, ..., v_{n_q}) \).
  • \( \text{eval}(n, e, \Gamma) = q(\text{eval}(n, e_1, \Gamma), ..., \text{eval}(n, e_{n_q}, \Gamma)) \).
• We therefore have $u \xrightarrow{\star} v$ for some $v \sqsubseteq K \text{eval}(n, e, \Gamma)$.

• $v \neq \text{INH}$.

Lemma $\omega^{\text{INH}}$: For any evaluable higher-order instance $n\#a$ in a state $\langle T, \Gamma \rangle$ with defining expression $e$, we can rewrite any $u$ where $u \sqsubseteq K \text{term}(n)$ to $v$ where $v \sqsubseteq K \Gamma[n\#a]$, via a non-empty rewrite sequence of rules in $\text{getRules}(P)$. Further if $a$’s type is of the same size as $K$, then $v$ is not $\text{INH}$.

At any state $\langle T, \Gamma \rangle$ where
- $t' \in T$,
- $n\#a \in \text{instances}(t')$,
- $\forall n'\#a' \in \text{dep}(n, e, \Gamma) . \Gamma[n'\#a'] \neq \bot$
- $u \sqsubseteq K \text{term}(n)$
we have
- $u \xrightarrow{\star} v$ where $v \sqsubseteq K \Gamma[n\#a]$
- $v \neq \text{INH}$ if $a$’s non-terminal is of the same size as $K$.

Proof: Proof is by induction on the number of higher-order instances evaluated.

• Let $\mathcal{L}(\text{symbol}(n)) = \mathcal{L}(K)$ for some $K \in NT$.
• By Lemma $\sigma^{\text{INH}}$, $u \xrightarrow{\tau} u'$ for some $u' \sqsubseteq K \text{eval}_\alpha(n, e, \Gamma)$ and $r \in \text{ruleRHSs}(p, e)$.
• Since $\forall r \in \text{ruleRHSs}(p, e) \cdot (p, r) \in \text{getRules}(P)$ by the definition of $\text{getRules}$, we have $u \xrightarrow{\star} u'$ for some $u' \sqsubseteq K \text{eval}_\alpha(n, e, \Gamma)$.
• If $\mathcal{L}(\text{type}_a(a)) = \mathcal{L}(K)$ then $u' \neq \text{INH}$.

Base Case:

• For the first higher-order instance evaluated, we have $\text{dep}(n, e, \Gamma) = \{ \}$.  

• We therefore trivially have
  $\forall n'\#a' \in \text{dep}(n, e, \Gamma) . (\Gamma[n'\#a'] \neq \bot$ and
  - for any $u' \sqsubseteq K \text{term}(n')$ we have $u' \xrightarrow{\star} v'$ for some $v' \sqsubseteq K \Gamma[n'\#a']$, and
  - if $\mathcal{L}(\text{type}_a(a')) = \mathcal{L}(K)$ then $v' \neq \text{INH}$).

• By Lemma $\sigma^{\text{INH}}$, $u' \xrightarrow{\star} v$ for some $v \sqsubseteq K \text{eval}(n, e, \Gamma)$.
Therefore \( u \rightarrow^+ v \) for some \( v \subseteq K \) \text{eval}(n, e, \Gamma).

- If \( \mathcal{L}(\text{type}_a(a)) = \mathcal{L}(K) \) then \( v \neq \text{INH} \).

**Inductive Case:**

- For a non-initial evaluatable higher-order instance, all required instances are higher-order.
- Therefore by the inductive assumption
  \[ \forall n' \# a' \in \text{dep}(n, e, \Gamma) . \ (\Gamma[n'\# a'] \neq \bot \text{ and} \]
  - for any \( u' \subseteq_K \text{term}(n') \) we have \( u' \rightarrow^+ v' \) for some \( v' \subseteq_K \Gamma[n'\# a'] \), and
  - if \( \mathcal{L}(\text{type}_a(a')) = \mathcal{L}(K) \) then \( v' \neq \text{INH} \).
- By Lemma \( \sigma^{\text{INH}} \), \( u' \rightarrow^* v \) for some \( v \subseteq_K \text{eval}(n, e, \Gamma) \).
- Therefore \( u \rightarrow^+ v \) for some \( v \subseteq_K \text{eval}(n, e, \Gamma) \).
- If \( \mathcal{L}(\text{type}_a(a)) = \mathcal{L}(K) \) then \( v \neq \text{INH} \).

**A.5.5 For Every Constant Tree Creation Sequence, There is a Rewrite Sequence of the Same Length**

Given a constant tree creation sequence, the sub-sequence that generate the pruned versions of each tree can be linked to generate a larger sequence that is at least as long as constant tree creation sequence.

**Lemma \( \rho^{\text{INH}} \):** For a constant tree creation sequence
\[
\langle t_0, \Gamma_0 \rangle, \langle t_1, \Gamma_1, n_1\#1_1 \rangle, \langle t_2, \Gamma_2, n_2\#1_2 \rangle, \ldots
\]
where for \( i > 0 \), \( \mathcal{L}(\text{symbol}(t_{i-1})) = \mathcal{L}(\text{symbol}(t_i)) \),

we can define a function \( R : N \rightarrow \text{Term} \) so that for \( i > 0 \), \( R(t_{i-1}) \rightarrow^+ R(t_i) \), i.e., there is a non-empty rewrite sequence from \( R(t_{i-1}) \) to \( R(t_i) \) of rules in \text{getRules}(P).

**Proof:** Proof is by construction.

Let \( \mathcal{L}(\text{symbol}(t_{i-1})) = \mathcal{L}(\text{symbol}(t_i)) = \mathcal{L}(K) \) for \( i > 0 \) for some \( K \in NT \).
We can define an \( R : N \rightarrow \text{Term} \) such that for \( i \geq 0 \)

- There exists a sub-term \( u_i \) of \( R(t_i) \) such that \( u_i \subseteq_K \text{term}(t_i) \).
- \( R(t_{i-1}) \Longrightarrow^+ R(t_i) \) if \( i > 0 \).

This would imply Lemma \( \rho^{INH} \).

We define \( R \) inductively such that these conditions hold as follows:

**Base Case:** \( i = 0 \)

- Let \( R(t_0) \) be \( \text{term}(t_0) \).
- \( \text{term}(t_0) = R(t_0) \) is trivially a sub-term of \( R(t_0) \) such that \( R(t_0) \subseteq_K R(t_0) \).

**Inductive Case:** \( i > 0 \)

- By the inductive assumption, there exists a sub-term \( u_i \) of \( R(t_i) \) such that \( u_i \subseteq_K \text{term}(t_i) \) and \( R(t_{i-1}) \Longrightarrow^+ R(t_i) \).
- We need to define \( R(t_{i+1}) \) such that it has a sub-term \( u_{i+1} \) where \( u_{i+1} \subseteq_K \text{term}(t_{i+1}) \) and \( R(t_i) \Longrightarrow^+ R(t_{i+1}) \).
- Since \( n_{i+1} \in \text{nodes}(t_i) \) and \( \mathcal{L}(\text{symbol}(n_{i+1})) = \mathcal{L}(\text{symbol}(t_i)) = \mathcal{L}(K) \), then for any \( u_i \subseteq_K \text{term}(t_i) \) there exists a sub-term \( u' \) of \( u_i \) (and hence of \( R(t_i) \)) such that \( u' \subseteq_K \text{term}(n_{i+1}) \).
- Since \( n_{i+1} \# 1_{i+1} \) is an evaluatable higher-order attribute instance, for any \( u' \subseteq_K \text{term}(n_{i+1}) \), we have \( u' \Longrightarrow^+ u_{i+1} \) for some \( u_{i+1} \subseteq_K \text{term}(t_{i+1}) \) where \( u_{i+1} \neq \text{INH} \), by Lemma \( \omega^{INH} \).
- Let \( R(t_{i+1}) \) 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 \( u_{i+1} \subseteq_K \text{term}(t_{i+1}) \) and \( R(t_i) \Longrightarrow^+ R(t_{i+1}) \).

Ordering non-terminals in this way therefore 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.
**Theorem II\text{\textit{\text{\textit{\textit{\textit{\textit{INH}}}}}}}:** If the rewrite rules in \texttt{getRules}(P) are terminating and the non-terminals can be ordered as required, there is no infinite constant tree creation sequence, and hence no improper evaluation sequence.

**Proof:** Proof is by contradiction.

- Assume a function \( L \) from \( NT \) to a finite total order that satisfies the conditions in \textbf{Defn. \( \Lambda \)}.
- Assume there is an improper evaluation sequence.
- By **Theorem I**, there is an infinite tree creation sequence.
- By **Lemma \( \kappa \)**, therefore, there is an infinite constant tree creation sequence \( \langle t_0, \Gamma_0 \rangle, \langle t_1, \Gamma_1, n_1#l_1 \rangle, \langle t_2, \Gamma_2, n_2#l_2 \rangle, \ldots \)
  where for \( i > 0 \), \( L(\text{symbol}(t_{i-1})) = L(\text{symbol}(t_i)) \).
- By **Lemma \( \rho^{INH} \)** therefore, we can define a function \( R \) so that for \( i > 0 \), \( R(t_{i-1}) \Rightarrow^+ R(t_i) \).
- Thus there is an infinite rewrite sequence \( R(t_0) \Rightarrow^+ R(t_1) \Rightarrow^+ R(t_2) \Rightarrow^+ \ldots \) of rules in \texttt{getRules}(P).
- Therefore, if there is an improper evaluation sequence, either the rules are non-terminating, or the non-terminals cannot be ordered.
- Thus if the rules are terminating and non-terminals can be ordered, there is no improper evaluation sequence.
References


