Reliably composable language extensions

A Dissertation

SUBMITTED TO THE FACULTY OF THE

UNIVERSITY OF MINNESOTA

BY

Ted Kaminski

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

OF

DOCTOR OF PHILOSOPHY

Eric Van Wyk

May, 2017

Acknowledgments

I am unlikely to do justice to everyone I should thank here.

I should begin by thanking my advisor, Eric Van Wyk, who has been a been a better mentor than I had even expected going in, and I had pretty optimistic expectations. I would also like to thank my thesis committee members: Gopalan Nadathur, Mike Whalen, and Tim Hunter. All of them helped provide valuable feedback in polishing this thesis. I also want to thank my ex-committee member, Wayne Richter, who retired before I finished this thing off.

I'd like to thank other students in this department, without whom grad school would hardly have been as excellent as it was: Nate Bird, Morten Warncke-Wang, Aaron Halfaker, Michael Ekstrand, Sarah McRoberts, Andy Exley, Hannah Miller, Sean Landman, Dane Coffey, Fedor Korsakov, and you know, tag yourselves. Sorry Reid. I'd also like to thank the colleagues I've had a chance to talk shop with, especially Tony Sloane and Jurgen Vinju.

I'd like to thank math friends, Shelley Kandola and Maggie Ewing, who provided some welcome diversions into things that aren't a part of this thesis, unfortunately. As for Minneapolis in general, best place on Earth, there are too many people to name, but Eli Cizewski-Robinson must appear here.

And finally, thank you Mary Katherine Southern, best friend, girlfriend, and colleague, for your support through most of grad school, for helping me make sense of things, and for brightening my life. And a final thank you to my family, for their support as well.

This material is based upon work partially supported by the National Science Foundation under Grant Nos. IIS 0905581, ACI 1047961, and CBET 1307089. I'd also like to acknowledge the support of Department of Education GAANN fellowships (through grants P200A060194 and P200A100195.) I was also partially supported in this work through DARPA and the US Air Force Research Lab under Contract No FA8650-10-C-7076, in conjunction with Adventium Labs.

Abstract

Many programming tasks are dramatically simpler when an appropriate domain-specific language can be used to accomplish them. These languages offer a variety of potential advantages, including programming at a higher level of abstraction, custom analyses specific to the problem domain, and the ability to generate very efficient code. But they also suffer many disadvantages as a result of their implementation techniques. Fully separate languages (such as YACC, or SQL) are quite flexible, but these are distinct monolithic entities and thus we are unable to draw on the features of several in combination to accomplish a single task. That is, we cannot compose their domain-specific features. "Embedded" DSLs (such as parsing combinators) accomplish something like a different language, but are actually implemented simply as libraries within a flexible host language. This approach allows different libraries to be imported and used together, enabling composition, but it is limited in analysis and translation capabilities by the host language they are embedded within. A promising combination of these two approaches is to allow a host language to be directly extended with new features (syntactic and semantic.) However, while there are plausible ways to attempt to compose language extensions, they can easily fail, making this approach unreliable. Previous methods of assuring reliable composition impose onerous restrictions, such as throwing out entirely the ability to introduce new analysis.

This thesis introduces *reliably composable language extensions* as a technique for the implementation of DSLs. This technique preserves most of the advantages of both separate and "embedded" DSLs. Unlike many prior approaches to language extension, this technique ensures composition of multiple language extensions will succeed, and preserves strong properties about the

behavior of the resulting composed compiler. We define an analysis on language extensions that guarantees the composition of several extensions will be well-defined, and we further define a set of testable properties that ensure the resulting compiler will behave as expected, along with a principle that assigns "blame" for bugs that may ultimately appear as a result of composition. Finally, to concretely compare our approach to our original goals for reliably composable language extension, we use these techniques to develop an extensible C compiler front-end, together with several example composable language extensions.

Contents

List of Figures

Chapter 1

Introduction

One of the ways we tame complexity is through abstraction. General purpose programming languages have evolved towards greater abstraction, from assembly languages, to the early procedural languages, to structured programming, object-oriented languages, and presently towards more functional programming. While the abstractions modern languages offer are a huge boon to programmers, there are at least two limitations with the current state of affairs.

The first problem is that the abstractions we have must frequently be broken, often to meet performance requirements. This is evident in very high-level languages, such as tuning Prolog programs for performance, or restating SQL queries until the optimizer decides to execute them in an efficient way. The correct function of these programs now depends (in an irritating and indirect way) on implementations details these abstractions were meant to hide. But this same problem also visible even in very low-level abstractions. As a simple example, after describing to novice programmers how to represent matrixes, we are quick to point out row-major order of arrays, CPU cache misses, and the performance impact of iteration order. Graphics programmers manually restructure arrays of objects (like point structures) into structures of arrays, to allow for SIMD vectorization. Java programmers today routinely make use of libraries that duplicate the implementations of various data structures for each primitive data type, despite the presence of generics, because the memory

and efficiency overheads of the generic versions are too high. In all these examples, performance concerns pierce the veil of abstraction and either result in abandoning the abstraction entirely (e.g. Java generics), or in creating a fragile and non-obvious dependence on those internal implementation details (e.g. SQL optimizer changes might break our queries). Regardless, the quality of the code suffers.

The second problem concerns the availability of abstractions. Many modern languages now include features baked directly into the language to support the use of features which actually appear in their standard libraries. One of the simplest examples is the "`foreach`" loop, which operates on an object that implements some sort of "`Iterable`" interface. Other examples include list comprehensions (in Haskell [1], or Python), special syntax for monads (in Haskell, or Scala [2]), and even a SQL-like query language (LINQ [3] in C#). These features themselves are not a problem—quite the opposite, as programmers seem generally quite happy to have these sorts of features. The problem is that these features must be baked into the host language, and consequently, they can only be built for things that are an integral part of the language, such as the standard library. Other libraries cannot introduce their own language features, to better support specific domains that would likely never be a part of a general purpose language. For example, it is unreasonable to expect the C standard committee to ever introduce a language feature to support the Python foreign function interface (FFI). C is expected to work on platforms Python will never appear on, and the Python FFI has changed in the past (and will likely change again in the future), and this would leave C with an obsolete piece of the language to maintain indefinitely. Despite the potential utility of such a language extension for some applications, we cannot reasonably expect it will ever be part of the standard language.

Together, these two problems paint a bleak picture, especially (to pick an illustrative example) for tackling the problems posed by heterogeneous or parallel hardware. Faced with processor technology that is approaching practical engineering limits for improvements to single threaded performance, many future significant increases in performance will have to come from making use of many processor cores of different types. But the task of writing software that can exploit multiple cores is complex and notoriously difficult. The complexity of parallel programming invites language abstractions to help manage, but the need to pierce abstractions for performance reasons make general purpose language design difficult. A large number of simple abstractions have been proposed that work well for certain narrow problem domains (MapReduce [4], for example,) but the inability to freely introduce new language abstractions means many of these never make it into general purpose languages. Unless there is an enormous breakthrough in parallel programming (e.g. we discover a "silver bullet"), it seems doubtful that general purpose programming languages will progress beyond a few simple utilities, like parallelizing maps over arrays. Utilities that, most likely, must be abandoned once performance begins to become too important.

1.1 Domain-specific languages

One hope for tackling these problems are domain-specific languages (DSLs). Instead of writing the entire program in a general purpose language, some parts of the program can be written in different DSLs, and so general purpose language designers do not need to address each individual problem domain. DSLs allow the use of higher-level abstractions appropriate for the problem domain, and also permit optimizations specific to that domain that may not be possible (or desirable) for a more general

```
exp:
  ...
  | exp '+' term   { $$ = mkAddExp($1, $3); }
  | exp '-' term   { $$ = mkSubExp($1, $3); }
```

Figure 1.1: A fragment of a YACC/Bison grammar. YACC is an external domain-specific language itself, besides also being a tool used to help implement many DSLs.

setting [5]. A classic example of a DSL is the YACC [6] parser generator, where the program is written at the high level of a context-free grammar (a small example shown in figure 1.1), and the resulting parser can be faster than a typical hand written one.

There are some disadvantages to DSLs, however. This particular kind of *external* DSL is a fully separate programming language, complete with its own set of tools and compiler. Using an external DSL typically involves confining a piece of domain-specific code to a separate file, and changing the project's build process to invoke that DSL's compiler first, which generates code that is then compiled along with the rest of the project. This approach limits the suitability of external DSLs to some degree. For example, it is hard to imagine what a useful external DSL for LINQ (as it appear in C#) or `foreach` loops would look like, as these are used in conjunction with lexically local variables nested inside other code. More importantly, external DSLs cannot be composed. Even if it were possible to make language features like `foreach` loops and LINQ into external DSLs in a reasonable way, if we want to use LINQ inside a `foreach` loop, we are simply out of luck with the external approach. They would be two separate languages.

In attempt to mitigate these problems, some external DSLs are embedded as strings in the larger program. Regular expressions and SQL queries are commonly constructed as strings at runtime and are then interpreted or compiled. But this

```
lazy val exp : PackratParser[Expression] =
  exp ~ ("+" ~> term) ^^ AddExp |
  exp ~ ("-" ~> term) ^^ SubExp |
  ...
```

Figure 1.2: A fragment of a grammar using Kiama [9] or Scala parser combinators. Following `^^` is a function to be called with the `Expression`s from `exp` and `term` that appear to the left of the operator. The `~>` operator causes its left operand to be parsed, but discarded.

approach has its own drawbacks, as it not only fails to detect errors until runtime, it also may not detect errors until tested with the right data. As a result, we have a pervasive problem with SQL-injection vulnerabilities, as just one example.

Internal (or embedded [7]) DSLs offer a solution to these particular problems. Internal DSLs are essentially ordinary libraries except that they take advantage of some host language features that allow them to approximate a more domain-specific syntax. A classic example of an internal DSL is a collection of parser combinators [8] (a small example shown in figure 1.2) that takes advantage of the ability to introduce or overload operators. Like any ordinary library, internal DSLs can be composed together and used by a programmer in the same way that several libraries can be imported and used. As a further advantage, internal DSLs naturally re-use the host language, avoiding an occasional problem with external DSLs, where they must sometimes re-implement a custom expression language, or accept host language code as unparsed strings. In the example in figure 1.2, a type error (such as accidentally using `CastExp` instead of `AddExp`, where the former expects different types) will result in an error reported in the user's code. Because the YACC example of an external DSL in figure 1.1 treats C code as unparsed strings, the type error would be found by the compiler consuming the generated code, not in the original code written by the user.

However, internal DSLs lack some of the advantages that external DSLs offer. Beyond restricted syntax, they are often limited in their ability to perform domain-specific optimizations or analyses by the capabilities of the host language. The lack of optimization is significant because it makes them more likely to be the kind of abstractions that a programmer may need to eventually discard for performance reasons. The lack of analyses has enormous consequences for how difficult they are to use, as useful properties cannot be enforced, the quality of error messages degrades, and the user may be forced to understand internal implementation details to interpret those errors. Many internal DSLs perform some kinds of analysis through a (usually complex) embedding of constraints into the host language's type system, for example in [10]. But the very complexity of these embeddings makes the error message situation go from bad to worse: the errors start to look like they're not just about a different program than the one the user wrote but appear to be for a different programming language, too. Parsing combinator libraries universally (to our knowledge) choose to permit only *ordered choice* of productions in the grammar being specified (that is, always preferring the first possibility before considering the second), because to allow unordered choice would raise the possibility of ambiguity, which they cannot reasonably detect nor report comprehensibly.

Comparing and contrasting parser generators and parser combinators is actually an exemplar of the differences. The former are able to generate efficient parsers, and raise errors for ambiguities in the grammar, but are uncomposable with other extensions, and treat semantic actions (written in the host language) simply as strings. Parser combinators are quite the opposite; for example, they often allow ordinary user-defined functions to generate rules in the grammar. But the resulting performance is relatively poor, and their formalism makes sacrifices (like lack of unordered

choice) to get around the lack of static analysis. These observations about efficiency, analysis, and composition generalize well to external and internal DSLs broadly.

1.2 Language composition

We would like to have all the advantages of external DSLs *and* internal DSLs. The central problem for external DSLs could be solved if we were somehow able to compose several DSLs together with a host language, but since these are separate languages, it's not clear how that should work. To be more precise about what we mean by language composition, we will use some of the classification and notation of Erdweg, Giarrusso, and Rendel [11]. This notation is abstract–that is, it is independent of any concrete mechanisms used to describe, implement, or compose languages.

To denote a host language (H) augmented with an extension specially built for that host language, we will use the notation $H \triangleleft E$. In this notation, H is a language and $H \triangleleft E$ is an extended language, but E is not considered a language by itself. In fact, although the use of the $\triangleleft$ operator would imply it, in the abstract E may not be any independent object in its own right. We might obtain $H \triangleleft E$ by taking a copy of H and directly modifying it.

Alternatively, to denote the composition of two full languages L_1 and L_2 we will use the notation $L_1 \uplus_g L_2$, where g is the "glue code" necessary to perform the composition. Here we have three known languages: L_1, L_2 and $L_1 \uplus_g L_2$, though again we have a rather under-specified component (g) that we merely assert is required for this composition. This glue code (g) is necessary to indicate how the composition should happen. Even when considering only the context-free grammar, we need to at least choose a start nonterminal for the composed language, and composition is

not really sensible without at least one new production that allows a transition from one language to the other. The glue code may also be necessary to repair any errors (such as ambiguities in the grammar) that arise after composition.

To permit only the $\triangleleft$ form of language composition ("language extension") is not sufficient for our purpose of trying to compose several DSLs. With H, $H \triangleleft E_1$, and $H \triangleleft E_2$, we are left with no option for composing all three, without rewriting one of the extensions to extend the other ($(H \triangleleft E_1) \triangleleft E_2$, for example). However, also including the $\uplus_g$ form of language composition still does not give us the necessary properties due to the need for glue code. Our goal is to support an ecosystem of many independent extensions, where the programmer writing an application can choose a subset. This user is not a compiler engineer and cannot be expected to develop the necessary glue code themselves. There is little hope of supplying this glue code, as there are not only a exponential number of subsets of extensions, but they are also developed independently from each other.

1.2.1 Composable language extensions

What we want is a way to automatically generate all the glue code necessary for composing languages. This way, the tools can take care of composition, and the exponential number of possible compositions becomes irrelevant (only mattering if we were trying to anticipate any combination the user might want). There is some hope of this if we limit ourselves to composing language *extensions*. For independently developed language extensions (giving us two extended languages, $H \triangleleft E_1$ and $H \triangleleft E_2$), we need glue code (g) to give us the composition $((H \triangleleft E_1) \uplus_g (H \triangleleft E_2))$ which composes the two. However, this should be the same language as $H \triangleleft (E_1 \uplus_g E_2)$, where we can see the glue code seems to be limited to reconciling the two extensions (and not a

more broad problem of reconciling two separate languages). This reduces our problem down to composing the extensions themselves.

Our goal, then, is to find a way to automatically compose extensions without needing any glue code (or more precisely by ensuring all glue code can be implicit and automatically generated). That is, we should always be able to obtain $H \triangleleft (E_1 \uplus_{\emptyset} E_2)$, where g is replaced by $\emptyset$, indicating no manually-written glue code is necessary to realize the composed language. However, this alone is not enough to be interesting. We can meet this property in a degenerate case where "extensions" are restricted such that they can change nothing about the host language at all. So we must introduce some lower bounds to the system's capabilities.

We define a language extension framework that is capable of *reliably composable language extension* as one that meets the following requirements:

1. Extensions may introduce new custom syntax, and may re-use the host language syntax.

2. Extensions may introduce new static analysis on both existing and new syntax.

3. Extensions are capable of generating informative, domain-specific error messages.

4. Extensions are capable of complex translation, such as domain-specific optimizations.

5. Given two extensions $H \triangleleft E_1$ and $H \triangleleft E_2$ we are always able to automatically obtain $H \triangleleft (E_1 \uplus_{\emptyset} E_2)$. That is, there must be no manually-supplied glue code necessary for composition to succeed.

6. The intended properties of each language H, $H \triangleleft E_1$, and $H \triangleleft E_2$ should hold for the result of composition $(H \triangleleft (E_1 \uplus_{\emptyset} E_2))$ as well. Or in other words, not only must the composition succeed, but the resulting language must work.

These goals encompass roughly the advantages of both internal and external DSLs as well as our desired composition property. Our requirement that we are able to provide good error messages (and relatedly, write new analysis) for extensions is a deeper requirement than it might appear at first. Abstractions do not simply allow us to avoid repeating common patterns, they also permit us to use additional properties in reasoning about programs. Error messages are the user interface that language abstractions use to ensure that code is well-formed. Without custom analysis and error messages, we would be almost certain to "leak" implementation details about an extension to the user, greatly complicating the use of an extension (for instance, because the user must now understand all those implementation details to understand the error messages).

Achieving all of these goals is a difficult task, and a conflict arises even with just the first two goals alone. Although external DSLs can have arbitrary syntax and analyses, our new goal is to *extend* an existing language with new syntax and analyses in a modular way. This turns out to be such a difficult problem, with substantial enough interest, that it has a name–the *expression problem* [12]. The central difficulty is the representation of abstract syntax trees. If represented in a typically object-oriented way, then we cannot add new analysis as an extension (because we'd have to modify an interface in the host implementation to add a new method). If represented in a typically functional way (as algebraic datatypes), then we cannot add new syntax as an extension (because we'd have to modify the host to add a new constructor to the

datatype).

Worse, as soon as we add the need to compose two such extensions together, the expression problem becomes the *independent extensibility problem* [13]. When the original language is extended with new syntax, and again by another extension with a new analysis, we must now somehow define that new analysis on that new syntax when we compose those two extensions. This is another of the ways in we might need "glue code" to get language extensions to successfully compose.

All of the above requirements together conspire to make this problem more difficult. Without the need for new syntax, a compiler written in a functional language would suffice. Similarly, without the need for new analysis, an object-oriented compiler would suffice. Without the errors and translation requirements (and arguably other requirements too), we have internal DSLs. Without the composition requirements, an external DSL would do. As we will see much later, a macro system could satisfy some of these requirements, by sacrificing either successful composition, or new analysis. And without the final two requirements for automatic composition, just a solution to the expression and independent extensibility problem should suffice. Together, however, no existing means of implementing languages or language extensions suffices, as we will see in the next chapter. (This paragraph is actually summarized visually later on, in section 2.3 on related work.)

1.3 Motivation for composable language extension

We began with two observations about the current state of general purpose language design. First, abstractions provided by current languages sometimes need to be broken (usually for performance reasons,) and second, language features are limited to

only those general-purpose enough to make it into the standard. Composable language extensions offer solutions to both of these problems.

Language extension liberates programming language design by making it possible to build features that are only useful to small portions of the community and not just the most general-purpose features. Instead of introducing language support for working with (as examples) the foreign function interfaces of Python, Lua, Matlab, and so on into standard C, each of these can be supported by a language extension. The standards committee need not be concerned with them.

Having a wider library of language extensions also helps us with the need to break abstractions. At issue is the idea of semantic mismatch: the language provides us with a mechanism, but often this mechanism is over-specified, and as a result is not flexible enough to accomplish what we want. (For example, we don't get "a matrix" we get "a row-major two dimensional array.") This problem can potentially be tackled in two different ways.

In some cases, this problem is a deficiency in the language, but usually has a well known work-around or pattern. For representing matrices, as a simple example, we simply need to make sure iterations over matrices are done in the most efficient order. Language extensions offer a way to codify (by either syntactic sugar or analysis) the use of these patterns. However, any particular abstraction design usually has trade-offs, and this often precludes including these into the host language. (There are many different matrix representations, for example, and host language design often stops after trying to include some operator overloading functionality to support them in a limited way as libraries.)

In other cases, an ordinary program with the usual language semantics could be mechanistically transformed into something more efficient [14], but this transforma-

tion is simply far too specific to apply generally. For instance, there is a large body of research on how to automatically parallelize a "for" loop, but little of it has impacted widely-used compilers, as the costs of integrating into a production compiler and running an optimization can exceed the perceived benefit[15]. Language extensions offer a way to incorporate these only when desired, without imposing costs on those to whom these features are irrelevant.

In additional to offering potential solutions to these problems, we get more opportunities as well. While our need to break abstractions for performance reasons is sometimes due to over-specified features, languages often leave us without high-level features (possibly in part because the standards do not wish to then be stuck having mistakenly over-specified them). This means, in C for example, we are given only structs, enums, and unions, and we're on our own if we wish to work at the level of objects or algebraic datatypes. Language extensions offer us the ability to create these language abstractions, and have the above benefit of being able to swap them out for the one with the right implementation details, should that later matter.

Lastly, *reliably composable* language extensions should enable an ecosystem of different solutions to problems. As a focused example, we can look to Haskell, where the GHC compiler has several different parallel programming techniques integrated, including Parallel Haskell [16], Concurrent Haskell [17], Data-Parallel Haskell [18], and software transactional memory [19]. Together, these have made Haskell an interesting playground for parallel programming research. But this is possible only because each of these is integrated into the monolithic compiler. Reliably composable language extension would permit these to be developed modularly, by independent parties, without the need for a central authority to decide what will be allowed in or not.

Empowering independent third parties puts language features on the same level

as libraries. Programmers are already accustomed to making decisions about what libraries are reasonable to depend upon. An ecosystem of language extensions allows them to make decisions about which language features are reasonable on a program-by-program basis. This frees the language standard committee from an impossible task of trying to please everyone, while giving everyone much of what they want. As we will see much later, it does not free language designers from having to design or evolve their host language, but it does focus their task. New features in the host language can enable new language extensions that might not otherwise be possible, and the standard library may wish to evolve to take advantage of new features as well.

1.4 Contributions

This thesis makes three major contributions, as well as two more minor ones that merit attention here as well.

Our chosen setting for investigating language extensions is attribute grammars [20], which are a kind of programming language well-suited to implementing compilers. Attribute grammars with forwarding [21] offer a uniquely promising approach to composing together language extensions. We can take a host language implementation, and several language extension implementations for that host language and attempt to compose them in a straight-forward way. Forwarding is the key reason why we're interested in attribute grammars specifically, everything else about attribute grammars has close analogs in other kinds of programming languages. (These concepts will be explained more in the background of chapter 3.) Our first minor contribution is to define (in chapter4) an attribute grammar-based programming language Ag, with a

useful combination of language features well-suited to compiler implementation, plus the addition of a novel pattern matching semantics that does not compromise the extensibility of the language being implemented.

Although this language has useful features for implementing composable language extensions, it does not solve the problems associated with composition. For instance, it is perfectly possible to implement extensions that work in isolation, but as soon as they are composed together cause the host compiler to crash due to missing glue code. In chapter 5, we contribute a *modular well-definedness analysis* for Ag. This analysis, for attribute grammars with forwarding, restricts what language extensions are capable of slightly, in order to guarantee that composition of extensions will always succeed without any missing glue code. This ensures that the composed attribute grammar will be well-defined (again, a concept we will describe in chapter 3) but in a *modular* way: we never examine the composed attribute grammar, only the host and each extension in isolation. This ensures the burden of making language extensions composable falls on the extension developers and not the user who is just trying to use them.

Next, in chapter 6, we introduce the problem of *interference*. Although the modular well-definedness analysis ensures no glue code is outright missing, there are still potentially problems. For instance, there is the question of whether the generated implicit glue code is the correct behavior in all cases. Or, we could be concerned that one extension computes values that flow into another extension where they are totally unexpected, causing misbehavior. We ensure problems of this sort are impossible, by turning the problem on its head and contributing a notion we call *coherence*. Coherence ensures that we are able to prove properties about our language extensions in a modular way, and we can compose these modular proofs and specifications success-

fully just as we can compose the attribute grammars. In some sense, this is the same composition problem but up a level, from attribute grammars to logic, and coherence plays a similar role to forwarding and the modular well-definedness analysis. Coherence does have a draw back, however: in order to ensure we preserve properties, all our properties must be coherent. This is a restriction on the kinds of extensions we can write, since we are then ruling out extensions that need incoherent properties to ensure their correctness. Coherence has a major advantage, too: it turns out there are effective methods for testing for inherent incoherence in attribute grammars, without having to do any verification or proof work at all. The final result are extensions we can be confident behave as specified when composed together, and so we have ruled out interfering misbehavior.

As our final major contribution, we synthesize all this work together to implement AbleC, a C compiler front-end that supports reliable composition of language extensions. This tool relies on the modular well-definedness analysis, the coherence approach to non-interference, as well as a "modular determinism analysis" for reliably composing syntactic extensions (this last from prior work, and introduced in section 3.3). This serves as an evaluation of these techniques, as well as a platform for further research.

As our last minor contribution, we observe that the combination of these three tools for ensuring reliable composition has interesting implications for language design. First, it defines a precise space of potential extensions around a host language. Extensions within this space are all reliably composable with each other, and outside they are not permitted by this system. The extent of this space depends on the host language and its design. And so, this also serves as a tool for helping host language designers focus on what is important. Instead of only being concerned with directly

integrating a language feature of interest, host language designers can be concerned with modifying the host language design to expand the space of possible extensions to encompass that feature instead. This can help focus design attention on important things, instead of on bikeshedding.

1.5 Outline of the thesis

In chapter 2 (related work), we look at some of the history of research into extensible languages. We discuss the expression problem in more detail, and survey the compiler construction and language implementation literature, discussing other work that is not directly drawn upon as part of our solution, and how they compare to it. Note that while we give an overall set of related work here, each chapter also includes its own concluding related work, and the related work of chapter 7 (section 7.7) is also relevant to the thesis as a whole, as this chapter is a synthesis of our contributions.

In chapter 3 (background), we introduce the prior work that we do directly draw upon for this thesis, in particular attribute grammars, forwarding, the Silver programming language, and the Copper parser generator [22]. Copper is an integral part of our solution, solving the syntax side of the composition problem, while this thesis solves the semantics side. Although this thesis is concerned with the Silver programming language, it is worth emphasizing that this language is for *implementing* extensible compilers. The languages that we are making extensible are any language with a compiler implemented using Silver. Silver is not the language we are interested in making extensible (although it is, by virtue of being implemented in itself).

In chapter 4, we introduce the language AG, which is the relevant subset of the Silver programming language. This integrates the various features of attribute gram-

mars we introduced in chapter 3, together with specifying its type system. We further develop a novel semantics for pattern matching, which ensures that pattern matching and attribute evaluation behave identically. This ensures that the use of pattern matching does not compromise the extensibility of the program (which forwarding was introduced to enable). Finally, we observe that, with Ag, the language composition operators merely specify which modules are being composed, and there is no special meaning to their shape, other than implications for dependencies between modules.

In chapter 5, we develop the *modular well-definedness analysis* that applies to Ag modules independently of each other. Full details of the previous chapter are not strictly necessary, though one should have a good understanding of how the language Ag works, as this is an analysis over it. This analysis is central to ensuring that composition of Silver modules will succeed, and as a result ensuring composition of language extensions will succeed. The primary restriction is that language extensions cannot introduce new non-forwarding productions for existing nonterminals, thus ensuring everyone agrees on a "ground truth" set of these. The remaining restrictions are primarily concerned with other aspects of ensuring attribute grammars are well-defined, including the use of *flow types* for tracking dependencies.

In chapter 6, we introduce the problem of *interference*. With the exception of understanding how non-forwarding productions are special (as well as understanding how forwarding works generally), this chapter does not require in-depth understanding of previous ones. (With the possible exception of a good understanding of decorated trees, emphasized in section 4.2.1.) To ensure extensions are non-interfering, we impose a requirement that they only rely on *coherent* properties. Given this, we show how we are able to take modular proofs of properties and extend these auto-

matically to the composed attribute grammar, ruling out interference. Further, we show that there is a testing-based approach to finding interference problems, without requiring the extension developers to write specifications and proofs of their extensions. Despite the potential for testing to miss bugs, coherence also gives us precise notion of blame: if a interference problem does arise in practice, it will be the fault of a single extension observable in isolation, and not a "gestalt" failure arising from the composition.

In chapter 7, we synthesize these tools into AbleC, a C compiler front-end capable of reliable composition of language extensions. For AbleC, we build several example language extensions under the constraints imposed by our analyses, and we attempt to probe the upper and lower bounds of the space of extensions we can build for AbleC. We also observe how changes to the C host language (and AbleC host language implementation) could have enabled a broader class of language extensions.

Finally, in chapter 8, we re-summarize these contributions with more detail, note important future work, and conclude.

Chapter 2

Related work

Before we move on to background on attribute grammars and the contributions of this thesis, we take a look at other language specification systems. Many of these are simply methods of building compilers, but each has different characteristics. At its simplest, for example, if we choose to represent our abstract syntax trees (ASTs) using objects, then the compiler has extensible syntax (ignoring parsing concerns,) since new derived classes can be introduced. Or if the AST is represented using something like algebraic datatypes, then the compiler has extensible semantics, as arbitrary new functions can be written over the AST.

Beyond just looking at these tools as ways of specifying compilers and languages, there are also many tools that were created specifically to support extensible languages. In this chapter, we will explore how each of these does not meet our goals. In some cases, there is no support for *composing* language extensions. In others, the kinds of language extensions that can be composed are far too limited (do not support new analysis, for example.) Finally, there are some which can attempt to compose extensions, but without any assurance of success. The result of composing several extensions together may be a compiler that crashes on some inputs.

In this chapter we take a very high-level view of each of these systems, and compare them primarily to our goals. In later chapters, as we make specific contributions, we will make more detailed comparisons where it is interesting to do so.

2.1 Extensible languages, historically

The notion of extensible languages is not a new one, it dates back to nearly the invention of compilers themselves. However, early work focused on the idea of an *extensible language* rather than *language extensions* more generally. As a direct consequence, most of the work focused on *macros* as the tool to achieve their goals, sometimes to the point of there being nothing but macros between parser and assembler (for example, META II [23].) These were often quite rudimentary; Algol 60, for example, simply had call by name procedures. More modern macro systems will be examined more closely in section 2.3.4.

One of the earliest pieces of relevant historical work was the introduction of the notion, not of an extensible language, but of an *extensible compiler*. In 1971, Scowen wrote [24]:

> "The normal approach in providing an extensible programming language seems to be to design and implement a base language which has facilities enabling the programmer to define and use extensions. This paper discusses a solution using an alternative approach in which extensions are made by changing the compiler."

They defined several goals for an extensible compiler which, in essence, covered most of our goals: new syntax, new semantics, good error messages, and complex translation.

However, there are two notable things about these goals that well characterize the historical research in extensible languages. First, the possibility of composing together multiple extensions was not considered. Language extension was considered just a way to modify a language to better suit some purpose. Second, one of the

stated goals for their approach requires that "it is possible to define a subset or to change the meaning of a language." This goal is in direct conflict with composition, and may be why composition was never considered a possibility. If one extension unexpectedly changes the meaning of the host language, we cannot realistically expect other extensions to continue to work. As a result, the goals of much early language extension research differed considerably from our own.

Another revealing piece of historical work is a paper introducing Simula 67 to the International Conference on Extensible Programming Languages in 1971. This conference took place twice, representing the peak of historical interest in extensible programming languages, and it's worth consider why interest died down subsequently. Simula was described–unexpectedly, to us–as an extensible programming language [25]:

> "Extensions can be divided into syntactic extensions[1] and extensions by introduction of new data types and of operations on these data types. The extensions in Simula are of the latter form. They correspond to the concept of class."

We believe this hints at, essentially, a profoundly different reason for interest in extensible languages than modern reasons. Programming languages of this era lacked sufficient abstraction mechanisms. The problem of introducing a new type, like complex numbers, or linked lists, looked like a language extension problem, because extending the language was the only known way of introducing such things. With the development of object-orientation and other abstractions (e.g. data and procedural

[1]It is worth pointing out that new data types and operations would otherwise be syntactic extensions, so this seems more identifying a subgroup of syntactic extension rather than a separate group in contrast to it.

abstraction and modularity [26], and later algebraic datatypes [27] just to suggest a few), interest in language extension dried up. The primary need for a way to create new abstractions was better met in another way.

It is interesting to consider this history because we should ask the question: "what's different now?" Why isn't it enough to simply resort to modern abstraction mechanisms anymore? There are many possible explanations for why (and the introduction touches on some), but it is enough to simply observe that modern programming resorts to linguistic sorts of abstractions frequently—and suffers for our poor support of these things. This can range from the internal and external DSLs mentioned in the introduction, to complex program configuration tooling that uses reflection and XML, to (ab)use of crude language extension mechanisms like Java's annotations or Python's decorators. Our interest stems from the observation that nearly all of these would be greatly improved if they were implemented as proper language extensions, instead of cobbled together in whatever way seems least-bad. As a general rule, any problem which calls for meta-programming, or otherwise generating code, is likely one that would benefit from language extensions. (And if one wishes for a more direct answer to this question, we speculate about some "killer applications" of our work by the end of this thesis.)

2.2 The expression problem

One of the first obstacles we must tackle to support extensions to semantics and syntax is the representation problem for syntax trees. The two obvious approaches, objects and algebraic datatypes, are dual to each other with respect to extensibility. This duality is the basis of the expression problem [12], mentioned in the introduction.

The problem is much more general than just syntax trees, as it applies to data representation in general, and although it wasn't given a name until 1998, its observation extends back to at least 1975 [28]. The problem can be visualized as a table [29], with one axis containing the *variants* of a type, and the other axis the *operations* overs that type. Objects permit new variants (new subclasses) while fixing a set of operations (methods), and algebraic datatypes permit new operations (functions), while fixing the set of variants (constructors.) By replacing "variants" with "syntax" and "operations" with "semantics", we can immediately see its application to language extension.

A quick aside: the word semantics can be somewhat overloaded. Here, we refer to almost any computation over a syntax tree as "semantics." One might also call these an "analysis." A more narrow definition would lead one to believe that syntax has exactly one particular semantics (usually the operational or runtime behavior), however we use the word semantics more broadly. There's not just translation or interpretation, but type checking, other statically detectable error checking, analysis supporting other transformations (such as "is this a constant expression?") and so on. A language extension might introduce (among other things) a constant analysis to host language expressions that lacked one. This is what we refer to as a semantic extension.

There are several solutions to the expression problem, but that alone is not enough for us, however. To solve the expression problem, it suffices to require a linear ordering to how extension is done. That is, in the terms of the notation for language extension, merely allowing $((H \triangleleft E_1) \triangleleft E_2) \triangleleft E_3 ...$ is enough. The expression problem only requires that each successive extension could introduce new variants or operations, and is not limited to just one of these. It does not directly require the possibility

of composition. The independent extensibility problem [13] generalizes the expression problem further, by requiring that a solution deal with composing independent extensions. This requires at least $H \triangleleft (E_1 \uplus_g E_2)$ to be producible from $H \triangleleft E_1$ and $H \triangleleft E_2$. This relieves us from the need to modify the extensions, to base one on the language already extended with the other. The independent extensibility problem does not consider the need for glue code (g) to be a problem, and instead defines success as the type system recognizing and raising errors for missing glue code. There are far too many solutions to the (independently extensible) expression problem to catalog, but perhaps the most notable are object-algebras [30] (which we will see more of in section 2.3.3), which have the distinguishing feature that it's a solution that works in plain Java. Similar solutions exist for functional languages like Haskell [31], and even theorem proving environments like Coq [32].

2.3 Language specification systems

We survey a number of language specification systems. Many of these systems do not focus on language extension, but have some support nonetheless. They often view language extension merely as a nice way for language designers to re-use an existing implementation to create a new language, not as a feature a programmer (as opposed to a compiler developer) would want to take advantage of. Whenever a system does permit composition of extensions, we will take note of how, and in particular its limitations with respect to our goals. We group together these varying systems into categories in the following sections. These categories are roughly summarize in figure 2.1 according to the goals we outlined in the introduction.

Category	Sec	1:Syn	2:Sem	3:Err	4:Opt	5:Cmp	6:Cor	Notes
Objects	2.3.1	Yes	**No**	Yes	Yes	Yes	Yes	
Algebra	2.3.2	**No**	Yes	Yes	Yes	Yes	Yes	
Hybrid	2.3.3	Yes	Yes	Yes	Yes	**No**	?	Glue code required
Macros	2.3.4	Yes*	Yes*	Yes*	Yes	**No**	**No**	Other trade-offs possible.
Internal	2.3.5	Yes*	No*	**No**	No*	Yes	Yes	

Figure 2.1: Categories of domain-specific systems, in relation to our goals (section 1.2.1). Answers marked with asterisks have qualifications; see the relevant sections.

2.3.1 Objects

The MPS Meta-Programming System [33, 34] supports extension in a fashion similar to object-oriented programming. MPS avoids any issues with concrete syntax by making use of a *projectional editor* (inspired by Intentional Programming, which we will discuss in section 2.3.3) that operates on abstract syntax trees (ASTs) directly. Those ASTs are fundamentally represented in an object-oriented fashion. MPS allows the user to define *concepts* which correspond to classes, nonterminals, and productions. For example, one might define the concept *Expression*, and another concept *IfExpr* that extends the concept of *Expression*.

As a consequence of this design, MPS does support natural composition of purely syntactic extensions. However, introduction of new semantics to existing language concepts would require directly modifying, for example, the *Expression* concept, and this would break any sub-concepts in other extensions that do not appropriately implement the new analysis. Of course, entirely new concepts can be introduced with whatever semantics is desired, but this is different from changing existing ones.

Wyvern [35] does support *reliable* composition of independently developed (syntactic) language extensions, without abandoning parsing in favor of projectional editing. It uses a type-directed, white-space sensitive parsing technique that isolates program

fragments to be parsed separately by a language extension. This ensures extensions' syntax can be composed safely, but the approach does not accommodate introducing new analysis of the host language.

OMeta [36] is a system that embraces an object-like extensibility. Unlike our usual notion of object-oriented ASTs, where nonterminals and their inheriting productions are classes, OMeta treats *grammars* in a class-like way. Inheriting from a grammar allows new syntax to be introduced similarly to methods, producing a new grammar. OMeta allows tree traversal via pattern matching visitors, however, making its extensibility properties actually more like algebraic datatypes (in the new subsection.) OMeta lacks any mechanism for composing multiple extensions, however: there is no support for multiple inheritance of several grammars in to one.

2.3.2 Algebraic datatypes

Oddly enough, the most common examples of a algebraic datatype approach to compiler design are mainstream compilers (javac, clang) that are implemented in an object-oriented language like C++ or Java. These compilers are often fundamentally structured around something like the visitor pattern, which is a technique for emulating in an object-oriented language the algebraic datatype style of programming commonly found in functional languages. Consequently, these compilers are easily extended with new semantics (often new optimization passes or specialized static checkers), but any change to the language syntax is usually not possible without simply modifying the original compiler.

Also notable in this area is Kiama [9], which is an internal DSL for Scala that provides a programming abstraction similar to attribute grammars. (We will examine attribute grammars in more detail in section 3.1.) A limitation at least partially

caused by its status as an internal DSL, Kiama is not always able to extend the definitions of attributes to handle new syntax. Consequently, it is probably best employed on *closed case classes*, which are essentially Scala's notion of algebraic datatypes, making similar to traditional compilers with respect to extensibility.

2.3.3 Systems that have solved the expression problem

There are multiple related techniques for solving the expression problem in traditional programming languages. This includes "datatypes a la carte" [31], final tagless interpreters [37], and object algebras [30]. And there are some language tools using these approaches, such as Ensō [38]. Although these approaches permit composition, their support for it varies. In particular, each offers no direct mechanism for ensuring composition succeeds. Type errors are likely to result from attempting to compose extensions.

Another approach to solving this issue is to alter the language's data representation model, instead of trying to operate within it. Aspect-oriented programming [39], for example, starts with an object-oriented language and permits "aspects" to be developed separately from each other. Among many other features not relevant to us, this allows new methods to be introduced to a class along with implementations for each subclass. However, this too suffers from composition problems. The "weaving" phase (that reduces a program using aspects to an ordinary object-oriented program) may result in classes that do not implement some abstract methods as a result of composing several aspects. This is not a problem for the way aspect-oriented programming is usually used, as it can be solved by writing the appropriate glue code, but means this is not a solution to our problem.

JastAdd [40] is an attribute grammar-based system for implementing compilers.

While JastAdd has a heavily object-oriented flavor to its design (arbitrary Java classes can be "nonterminals"), it still fundamentally allows introduction of both new syntax and semantics. This is achieved through an aspect-like weaving phase during compilation. However, no attempt is made to ensure the safe composition of extensions, and in fact, failure is not even statically detected: if a new analysis can not be performed on new syntax, an error is raised at runtime.

UUAG [41] and AspectAG [10] are another attribute grammar-based system, but this time based on a functional programming language (Haskell.) Again, new semantics and new syntax can be introduced and composed, but composition may fail statically when it lacks an equation for some analysis on some production.

ASF+SDF [42, 43] (and systems building upon it, notably SugarJ [44]) and Stratego/XT [45] take an algebraic rewriting approach to language implementation. Both syntax and semantics can be introduced in extensions, but composition of extensions can have unintended behavior. Semantics are typically given by conditional rewrite rules, where the conditions make use of unification-based matching. Composition failure becomes rather unruly because rather than raising an error, a rewriting simply stops with an unintended term, which can then be obscured as it results in unification failure, which is easily conflated with whatever unification failure is intended to signify (for example, that two types are unequal.)

Rascal [46] is a spiritual successor to ASF+SDF, desiring an implementation method that is more like "just programming" than the rewriting approach. It is essentially a functional programming language in the vein of ML, but with a more Java-like syntax and a wealth of language processing features built-in. Rascal's module system is slightly unusual, but it does support extending both syntax and semantics of languages (data types in other modules), and spots composition problems

statically at composition time. Again, however, no attempt is made to ensure that composition of extensions will necessarily succeed.

Intentional Programming [47, 48] is the origin of the "projectional editor" approach to syntax that MPS also uses. Unlike MPS, however, Intentional Programming does not take an object-oriented approach, and consequently both syntax and semantics can be extended. Uniquely among the systems in this section, this approach shares many of our goals and does make an attempt to allow safe composition of extensions. "Questions" that are asked of nodes in a syntax tree are subject to "routing" in such a way that it's possible for extensions that introduce syntax to "answer questions" that are introduced in an independent extension. This work is the origin of forwarding in attribute grammars [21], which we will make use of later. However, the system has no mechanism to ensure all questions are always answerable, and so it ultimately makes no guarantee the composed compiler will work.

The Delite project uses Scala as a host language for domain-specific constructs and supports lightweight modular staging [49], a type-based approach to multi-stage programming, to analyze and optimize DSLs in Scala. Delite uses the embedded DSL approach to syntax extension, and comes with a solution to the expression problem. However, composition of several Delite DSLs may require glue code to accomplish. Further, the project is focused on optimization and transformation issues, and not on quality of error messages, which is another of our major goals.

2.3.4 Macros

Macro systems are worth special attention. Typically, macro systems cannot actually extend the "real" syntax of the language. Languages like Lisp that support macros have a fixed concrete syntax, S-expressions, which is flexible only in certain ways.

Lisp-like macros amount to extending the abstract syntax with new productions that give interpretations to some S-expr trees. Thus, macros systems are usually thought of as a syntax extension mechanism.

Somewhat similar to Lisp, C++ templates also allow for some macro-like behavior. Although more constrained syntactically, they are nonetheless capable of arbitrary computation. However, templates also notoriously suffer from extremely obtuse error messages that can span pages.

Perhaps the most interesting system with macros is Racket [50] which supports hygienic macros [51]. Interestingly, this system implements a type system on top of an underlying dynamic language through macros (called Typed Racket), demonstrating that their macro system is quite capable of semantic extensions and not just syntactic ones. At first glance, this seems to satisfy nearly all of our goals, as it can extend syntax, semantics, and do quite complex translation with good error messages. However, macros still suffer from composition problems. This is easily exposed in Typed Racket by attempting to define a macro that expands into a *type* instead of a term. This results in Typed Racket encountering a syntactic form it is not familiar with (as it inspects the ASTs prior to their macro expansion, so that it can give types to macros, too), and consequently fails to compile. And so although macros may introduce new syntax and inspect subtrees to introduce new semantics, they are still fundamentally unsafe to compose if one attempts to do both.

Ciao [52, 53] is a logic programming language with a language extension system similar to macros. One nice feature of Ciao is that its language extensions are simply libraries, and are scoped in their applicability. Ciao gains some nice composition benefits from its module system, as its rewrite rules on terms apply based on the module the names originate from. That is, a module can introduce a predicate p

and a rewrite rule for that predicate, and whether that rule fires on a p in the client program will depend on ordinary name resolution. If the p resolves to the p exported by that extension module, the rule applies, otherwise the rule is not applied. Although this helps avoid spurious conflicts between extensions, ultimately Ciao still has the same limitations as macro systems generally with respect to our goals. In particular, it is possible for extensions to conflict and not compose gracefully.

2.3.5 Common host languages for internal DSLs

Finally, we turn our attention to the salient features of languages suitable for implementing internal/embedded DSLs. These languages have features that "virtualize" syntax: that is, syntactic features of the language turn into mere library calls in some fashion, and there is usually some way to override the behavior of these calls. This can take the form of either static or dynamic dispatch based on types, for example. These languages frequently also allow the introduction of "unusual" kinds of symbols: perhaps new operators, or unusual method names (involving symbols), and so forth.

Haskell, for example, has several pieces of virtualized syntax, most notably monads and do-notation, but also including its numeric syntax (`+` and so on) as well. Haskell also permits introduction of brand new operators with arbitrary association and precedence rules. Scala is similar: it has special syntax for "flatMaps" which essentially correspond to do-notation, and method names in Scala can contain almost arbitrary characters (for example, `x + y` is an invocation of the `+` method on the object `x` with parameter `y`.) Scala also supports "implicit conversions" that only occur within certain scopes, which can allow more seamless use of host language objects, by automatically converting them to a form appropriate for the internal DSL. Both languages support generalized algebraic datatypes, with claimed applications

to internal DSLs, but their use in practice is less clear.

C++, in addition to supporting some macro-like features, also supports some internal DSL features. Operators on types can be overloaded, including unexpected ones like dereferencing and conversion operators, and although new operators cannot be introduced, old ones can be re-purposed. C++11 introduces additional virtualized syntax, like curly-brace initialization.

Ruby is also a somewhat popular language for internal DSLs thanks to its extremely dynamic behaviors. Most notably, even method calls are actually virtualized: if a method is called that lacks an implementation, a generic "method not found" method is called on that same object, that can be overridden to potentially provide an appropriate implementation! These are often used to, for example, automatically generate methods appropriate to a database table, based on runtime information about the table's schema.

All of these internal DSL systems primarily fail to provide efficient translations, usually do not have good error messages, and the syntactic extensions are limited to what virtualized syntax the host language provides. The nature of virtualized syntax is that it turns into mere library calls, and these library calls often simply construct trees that are then interpreted at runtime. Those that don't, must still contend with function call overheads that are purely an artifact of the syntax used, and not essential to the solution. To be efficient, these often require the mythical (and nonexistent) sufficiently smart compiler to perform exactly the right amount of partial evaluation.

The error checking situation is worse. While there is no static checking at all for dynamic languages like Ruby, the only real analysis possible for the static languages is via type system embeddings (e.g. those used in AspectAG [10]). However,

although this allows properties to be enforced by types, the error messages are often unreasonable, producing large difficult to understand errors about types the user is often unexposed to, much like C++ template error messages. Many internal DSLs sacrifice static checking in favor of dynamic checking via reflection and the use of test cases. This is a significant motivator for doing better than internal DSLs, as what an abstraction hides is almost as important as what it allows. The ability to analyze the correctness of code at the level in which it is written, instead of in terms of its implementation details, is an important part of making language extension useful.

Chapter 3

Background

Before we get to the contributions of this thesis, we will briefly introduce attribute grammars (AGs), some extensions to AGs, and two tools that are a core part of this work. These tools are Silver (as it existed prior to our contributions,) an attribute grammar-based programming language, and Copper, a context-aware parser generator. We will introduce attribute grammar concepts by example throughout this chapter. These examples will be written with Silver syntax, introducing both the concepts and the Silver language at the same time. This will hopefully serve as a partial Silver tutorial, though restricted to those concepts which are of importance for the purposes of this thesis. We will also refer back to some of these examples later in the thesis.

Our goal here is to build some intuition for what attribute grammars are and how they work. Formal definitions will not be presented here, though some will appear as needed later in the thesis. For example, we do not present a Silver grammar here, though a restricted subset of Silver is given a grammar in chapter 4. For our purposes, the formal definitions of some of these features will distract rather than enlighten.

Section 3.1 concerns attribute grammars, and their various extensions. Section 3.2 concerns Silver, and section 3.3 is about Copper. Readers who are already quite familiar with attribute grammars may wish to simply glance at figures 3.3 and 3.5 before skipping to section 3.3.

3.1 Attribute grammars

Attribute grammars [20] are a formalism for describing computations over any sort of tree structure, though in particular, syntax trees. These trees are constructed from an underlying context-free grammar by tradition. Though any tree structure (such as an algebraic datatype) will do, we continue this tradition of using a grammar and so describe our types as *nonterminals* and their constructors as *productions*. But most of the examples we will present are abstract syntaxes–we will use types like `Integer` as our "terminals" rather than proper syntactic terminals with associated regular expressions.

The nodes in these trees are then decorated with *attributes*, which can give "meaning" to each node in the tree. Generally, this is just a fancy way of saying that attributes at the root of a tree compute some function over that tree, and attributes in the middle of the tree represent essentially recursive application of this function on subtrees. Attributes are divided into *synthesized* and *inherited*, which indicate whether they make information flow (respectively) up or down the tree. Each production in the grammar comes with a set of *equations*. Equations for synthesized attributes define the value of that attribute for nodes in the tree corresponding to that production. Equations for inherited attributes define the values that should be handed down for each child of that production. These equations may depend on the values of other attributes on that same node and its children.

As a simple example of an attribute grammar (following Knuth's original example), consider defining a simple language for binary natural numbers. We will have a `Bit` nonterminal, which is either zero or one, and a `Nat` nonterminal, which is a list of bits. As "semantics" for this syntax, we wish to compute the actual integer value

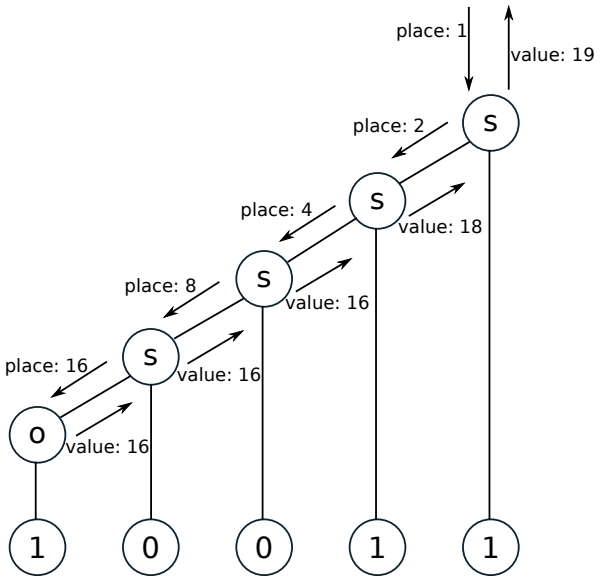


Figure 3.1: A visual example of an attribute grammar, showing 10011 represented as a `Nat` tree.

that a given tree represents, which we will call the synthesized attribute `value`. As part of computing this value, we communicate, going down the tree, what the place value of each bit is. In order to make this easier, we choose to represent `Nat` as a sort of "reversed list" where the top of the tree appends to the end[1], and thus we can always report the root's place value as 1.

In figure 3.1, we show an example tree, the binary string 10011. We have labeled the values of the attributes as they should flow down and up the tree. Note that we have not yet given the declarations and equations that define this attribute grammar, we are only showing an example of a decorated tree in this figure.

In figure 3.2, we show the definitions of the two nonterminals: `Bit` and `Nat`. We define a single synthesized attribute on `Bit` that simply translates the tree structure

[1]Traditionally, lists are constructed by adding elements to the front using a constructor called `cons`, short for constructor. And so to add elements to the end, we follow cute tradition and name this construct `snoc`.

```
inherited attribute place :: Integer;
synthesized attribute value :: Integer;
nonterminal Bit with value;

production one
top::Bit ::=
{
  top.value = 1;
}
production zero
top::Bit ::=
{
  top.value = 0;
}
```

```
nonterminal Nat with value, place;

production oneBit
top::Nat ::= l::Bit
{
 top.value = l.value * top.place;
}
production snocBit
top::Nat ::= i::Nat  l::Bit
{
  i.place = top.place * 2;
  top.value = i.value +
    l.value * top.place;
}
```

Figure 3.2: The attribute grammar that would animate the previous example tree.

into a numerical value. For `Bit`, this is quite trivial. For `Nat`, we give the `value` attribute its meaning in terms of (1) the value of the natural number to the left, if any, (2) the place value of this bit, and (3) the value of the bit. Finally, we have the `snocBit` production tell its left child that its place value is twice the present one.

Returning to the picture presented in figure 3.1, we should consider for a moment how evaluation is accomplished. Traditionally, we would analyze how attributes depend on each other, and produce a static evaluator that was certain to compute the needed values of attributes before they were demanded by the equations of others. Notably, ordered attribute grammars [54] would observe that we could evaluate this attribute grammar in a single pass: computing the values of `place` going down, and then the values of `value` returning back up the tree. However, we simply take attribute grammars to be lazily evaluated purely functional programs [55], instead of attempting to determine (eager) static evaluation schedules. That is, we fully decorate a tree with thunks, and accessing an attribute simply corresponds to demanding the

appropriate thunk. Either the thunk has been evaluated before and is already cached, or it will be evaluated immediately, possibly demanding more thunks corresponding to other attribute values that it depends upon, and so on. Attribute grammars can still be poorly formed: true (non-productive) circularities may be consider invalid, along with other evaluation difficulties such as missing equations. However, we do not further restrict them in the hopes of achieving a rigid evaluation order.

3.1.1 Higher-order attribute grammars

Higher-order attribute grammars [56] allow attributes to themselves contain trees that are as-yet undecorated by attributes. That is, we have so far seen attributes with types like `Integer`, but this would allow attributes to have nonterminal type, like `Nat`. These trees are made useful by permitting productions to "locally anchor" a tree and decorate it, as if it had been a child. A child, however, is supplied when a tree is created (i.e. as an argument when we invoke a production), whereas these "virtual children" are computed by an equation locally to a production and may depend on the values of attributes like any other equation. We introduce this "anchoring" operation by allowing productions to declare local variables, but treating specially local variables of nonterminal type. For these, we allow inherited attribute equations to be supplied as if they were child instead of a local.

Circular programs [57] are presented as a particular method of describing tree traversals in lazy functional languages. These are, in fact, just attribute grammars described without the benefit of attribute grammar notation. A circular program is generally described as performing a tree computation "in one pass." This is a somewhat misleading description, as people often believe this grants some performance benefit, but it is really about the semantics of the tree traversal. Without circular

```
synthesized attribute min :: Integer;
inherited attribute global :: Integer;
synthesized attribute rep :: RepMin;

nonterminal Root with rep;
nonterminal RepMin with
    min, global, rep;

production root
r::Root ::= e::RepMin
{
  r.rep = e.rep;

  e.global = e.min;
}
```

```
production leaf
e::RepMin ::= x::Integer
{
  e.min = x;
  e.rep = leaf(e.global);
}
production inner
e::RepMin ::= x::RepMin  y::RepMin
{
  e.min = min(x.min, y.min);
  e.rep = inner(x.rep, y.rep);

  x.global = e.global;
  y.global = e.global;
}
```

Figure 3.3: The classic RepMin circular program, written instead as a higher-order attribute grammar.

programs (or attribute grammars), in a multi-pass computation over a tree, there is a communications problem to be solved. One pass of a compiler might infer the types of all expressions, while a later pass might need those inferred types to compute error messages. Somehow, the type information on each expression node in the tree must be communicated from one pass to the next. This has many solutions (mutating tree nodes in place to add the communicated information, or constructing a new tree type which is then used by later passes, for example), but circular programs eliminate the problem entirely. The later "pass" (such as an `errors` synthesized attribute) can simply directly access information computer by an earlier pass (such as a `typerep` attribute). The information computed for a node deep in the tree is simply locally available no matter what "pass" is currently being computed. It is in this sense that the computation proceeds in "one pass," and this is the chief advantage of the circular program (or attribute grammar) style of programming.

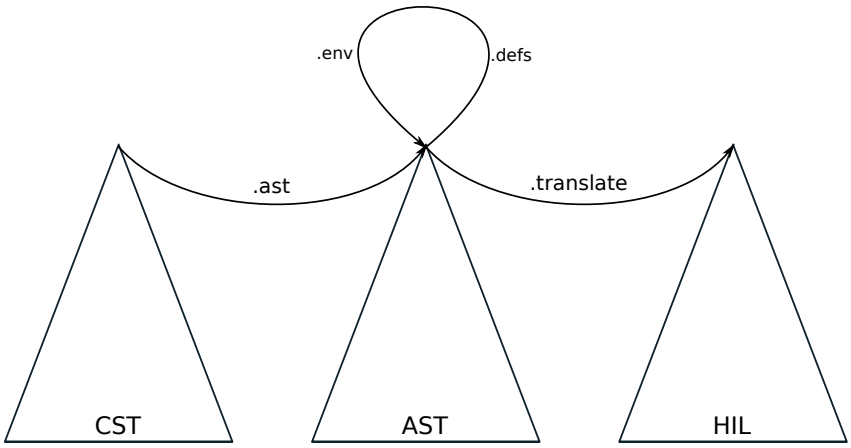


Figure 3.4: Tree transformation with higher-order attributes.

The classic example of a minimal circular program is the RepMin problem: taking a binary tree with numbers at the leaves, and replicating a tree structure of exactly the same shape, but with all the leaves replaced with the overall tree's global minimum value. (This classic example unfortunately does not display the above mentioned chief advantage in action, but instead merely shows all of the technical pieces involved in the construction of a circular program.) In figure 3.3, we show RepMin solved using higher-order attribute grammars. In the `root` production, we take the minimum computed from the subtree, pass it back down as an inherited attribute so that it can be used in `leaf`s to compute `rep`. The `rep` attribute is a "higher-order attribute" (or a "nonterminal attribute"), which constructs a tree. In the case of Silver these are just ordinary synthesized attributes, but with nonterminal type, and not really deserving of special attention themselves.

Higher-order attribute grammars are particularly useful for describing compilers, as they generalize tree transformations. Figure 3.4 shows the traditional compiler pipeline: concrete syntax to abstract syntax to (language-agnostic) high level inter-

mediate language. Higher-order attribute grammars, among other things, permit this standard architecture to be naturally expressed using attributes. Without this extension, the kinds of computations that can be performed are limited by the structure of the original tree the attribute grammar is being evaluated upon. Instead, we are able to transform trees into a representation more suitable for any particular calculation.

3.1.2 Forwarding

Forwarding [21] was introduced into attribute grammars to make it possible to compose language extensions, without running afoul of what is essentially the expression problem. A language extension that introduces new productions, combined with a separate extension that introduces new attributes on existing nonterminals, presents a serious problem: the new attribute may not have a defining equation on the new production. If these were introduced in independent extensions that were unaware of each other, then the composed AG would be incomplete.

Forwarding solves this problem by allowing extension productions to construct a *semantically equivalent* tree in the host language (that is, using only productions from the language they are extending). To accomplish this, we introduce a new kind of equation (besides defining synthesized or inherited attributes) that indicates what that production "`forwards to`." We call a production that has a forwarding equation a "forwarding production." When a synthesized attribute is requested from a forwarding production that does not have an explicit corresponding equation, the attribute can simply be evaluated on *forwarded-to* tree instead. For inherited attributes, we do this same automatic copy in the other direction. The forwarded-to tree is given a copy of the inherited attribute the forwarding node was given. As a result, we can generally be unconcerned about missing equations on forwarding productions, as we can obtain

a value via forwarding.

Although this behavior for forwarding appears at first to be syntactic sugar, the implicit copy equations forwarding would generate cannot be computed locally. That is, the implicit equations are a part of the final composed attribute grammar, the result of an extremely simple but whole-program analysis. This is why forwarding is able to help solve the expression problem, where is seems like syntactic sugar should not be able to: it is non-modular sugar. When one extension introduces a new (forwarding) production, and another extension introduces a new synthesized attribute, we need to generate the appropriate copy equation, but this equation cannot be placed in either extension if they are independent of each other. It must be generated (automatically, as one kind of "glue code") as a result of composition of these separate modules.

Forwarding is the primary reason we are interested in attribute grammars. Although it seems similar to macros at first glance, there are important differences. For one, we are able to write equations on forwarding productions (an example of which we will see shortly). This means that we could get semantics from what we forward to, but, for instance, give a value for the errors attribute specific to the forwarding production (instead of what we would have gotten from its expansion). Moreover, forwarding nodes in a tree are persistent: they are not implemented by being rewritten away, as macros are. This is a critical piece of how we ensure error messages can be about the code the user wrote, and not the internal implementation details of what it expands into.

In figure 3.5, we show a simple (abstract) grammar of boolean expressions. We can construct trees using `and`, `or`, `not`, and `literal` true or false values. From these tree, we can compute two attributes: `eval` gives us a boolean value that is the

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

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

Figure 3.5: An example grammar for boolean propositions, with two forwarding productions: `implies` and `iff`.

result of evaluating the tree, and `neg` (negation). This latter attribute transforms the tree into one which would give an opposite value by removing `not`s, applying De Morgan's laws (turning `and` nodes into `or` nodes with negated children, and vice versa), and swapping true/false literals. (These attributes are chosen semi-arbitrarily, as examples of things we might want to evaluate on a boolean expression tree.) To this little language we add two forwarding productions: `implies` and `iff` which correspond to their respective logical connectives.

There are three salient features to take note of. First, `implies` gives no equations for itself other than the forwarding equation. Next, `iff` does choose to give an equation for `eval` but not for `neg`. Finally, the forwarding equation for `iff` makes

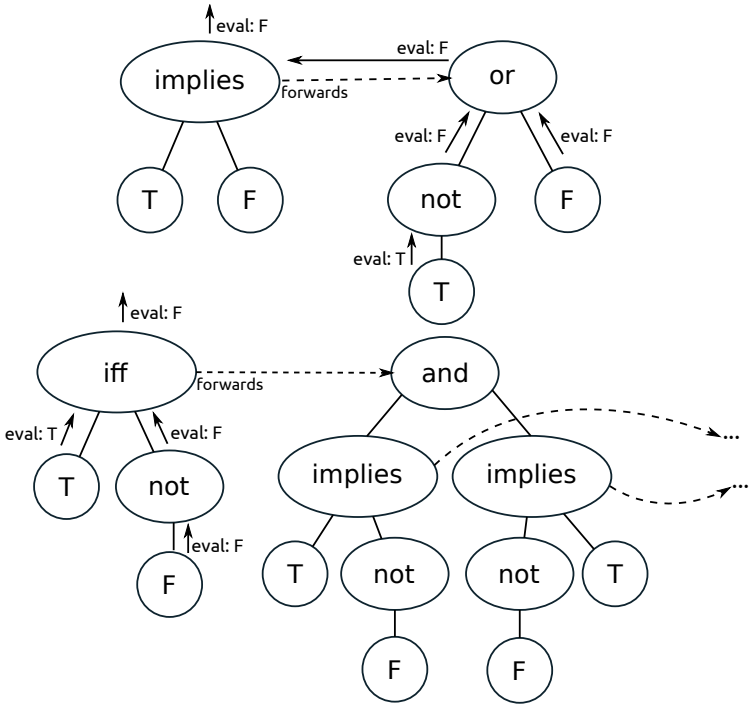


Figure 3.6: Example of two forwarding trees and the evaluation of the `eval` attribute.

use of `implies`. While we conceptualized forwarding as constructing an equivalent tree purely the host language, as long as the expansion of forwarding terminates, that is still (eventually) the case. There are just more indirections before we get to the purely host language tree.

In figure 3.6, we show two example trees, and visually explain features of how attribute evaluation and forward expansion proceed on decorated trees. In the top example, we see the `eval` attribute evaluated on an `implies` tree. Notice how the forwarded-to tree has `eval` computed on it, but also take note that we do not touch the children of `implies` at all. In the bottom example, with the `iff` production that gave an explicit equation, we see that `eval` is evaluated on its children, and the forwarded-to tree is not touched.

Finally, notice the duplication of the children in both of these examples, partic-

ularly the `not(F)` tree in the bottom example. We see it drawn three times, and in fact that is only because we have not drawn the next expansion of those `implies` productions, which would have given us a total of five instances. This is a necessary part of forwarding, as each of those locations in the tree can be given inherited attributes by its parent, and so in general each may have a different set of inherited attributes and may thus compute different values for synthesized attributes.

In the worst case, this can mean forwarding results in exponential tree expansion. However, in practice we often only see a constant factor expansion. Typically, an attribute equation is supplied up front by the original forwarding productions immediately, or it is only evaluated on the final tree of non-forwarding productions, which allows us to "skip" most of the intermediate forwarding nodes. In the top example, is would be as though the children of `implies` never exist on their own at all. And in the bottom example, we might never even compute the forward tree of the `iff` production at all, and even if we did, we'd likely never compute anything about the children of the two `implies` productions. However, it is possible for forwarding performance to become a concern if we need to access attributes from children in order to decide what to forward to, and then see those subtrees duplicated in what we forward to. (Consider, for example, forwarding based on the types of subexpressions.) However, there are ways of mitigating the problem in these cases by making use of "references," the subject of the next section.

3.1.3 Reference attribute grammars

Reference attributes [58] were introduced to handle non-local dependencies across a tree, and are often described as superimposing a graph structure on the syntax tree. For example, one can use reference attributes to obtain, at variable use sites, direct

```
nonterminal Expr with env;
inherited attribute env ::
    [(String, Decorated Binder)];

production let
top::Expr ::= b::Binder  e::Expr
{
  e.env = (b.name, b) :: top.env;
}
production var
top::Expr ::= v::String
{
  local ref :: Decorated Binder =
    lookup(v, top.env);
}

nonterminal Binder with name;
synthesized attribute name :: String;

production binding
top::Binder ::= v::String  e::Expr
{
  top.name = v;
}
```

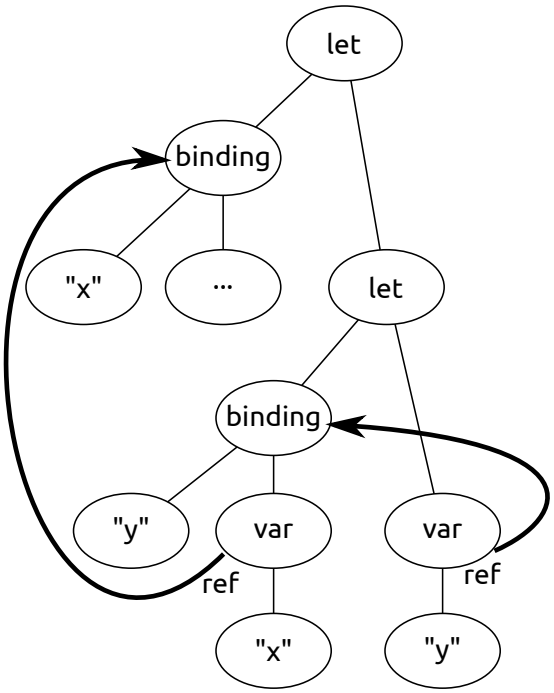


Figure 3.7: An attribute grammar and example tree showing references.

references to the declaration node for that variable. Similarly, one can augment an abstract syntax tree of statement nodes to each have references to all statements nodes that may precede or follow it, constructing a control-flow graph. References help specifications stay at a high-level, as the limitation to trees only can make some computations much more difficult to express.

Silver (idiosyncratically) introduces an explicit "decorate type" for each nonterminal in order to permit references. We will go into more detail on this in section 4.2.1. In figure 3.7, we can see part of an attribute grammar that passes a reference to the

declaration site of a variable, through the environment, to be looked up by the use site of a variable. We can see how this creates edges in the tree that did not exist previously (and thus is no longer really just a tree). Attributes can be accessed across these edges the same as if it had been a child (but inherited attributes cannot be supplied across these edges in Silver.) References refer to the existing attributed nodes in the tree, whereas higher-order attributes would have referred just to the basic tree structure. If a higher-order attribute had been passed along and locally anchored instead, we would see a new copy of the subtree, not a direct reference to where it already exists. Among other things, this means one can obtain (for example) the environment of the referenced node (whereas with higher-order attributes, we would have to supply an environment to the new copy of the subtree).

3.1.4 Aspects

One of the reasons we are interested in attribute grammars in the first place is that they have a good composition mechanism. We can take the sets of nonterminals, productions, attributes, and equations and just union them together. However, the syntax of Silver we have presented so far makes equations seem tightly coupled to production declarations. That is, we don't have a way of writing equations for a production separately from introducing a production.

This would be a significant restriction, as introducing attributes (and equations for existing productions) is one of the goals of our system: we wish to be able to introduce new semantic analysis on existing syntax. Fortunately, this is not a problem with the attribute grammar model. It is only an apparent problem with the syntax we use to declare productions.

Silver has *aspect* productions that allow us to write equations separately from

```
synthesized attribute nots :: Integer;
attribute nots occurs on Expr;

aspect and
e::Expr ::= a::Expr  b::Expr
{
  e.nots = a.nots + b.nots;
}
aspect or
e::Expr ::= a::Expr  b::Expr
{
  e.nots = e.nots + b.nots;
}
```

```
aspect not
e::Expr ::= a::Expr
{
  e.nots = 1 + a.nots;
}
aspect literal
e::Expr ::= b::Boolean
{
  e.nots = 0;
}
```

Figure 3.8: Introducing a new attribute to the boolean grammar of figure 3.5 using aspect productions.

declaring a production. In figure 3.8, we extend the boolean expression language from figure 3.5 with a new attribute. This `nots` attribute simply counts the number of instances of the `not` production in the tree. Unlike traditional aspect-oriented programming [39], these aspects are extremely limited in capability. They are capable of having no non-local effects, as we can only introduce equations for new attributes, not modify or otherwise influence the behavior of existing attributes with existing equations. It is this difficulty in reasoning about program behavior that lead to problems with aspects. This more limited incarnation in Silver instead only leads to the expression problem we have already discussed. If we introduce an attribute and aspect every production we know about, what about the productions we don't know about? Well, that was what we introduced forwarding for in the last subsection.

Silver uses this aspect syntax to help keep equations for each production grouped together, and to keep name binding local. That is, if we allowed equations as top-level declarations, they'd be more likely to get scattered around without organization, and

they would be referencing children using names from some production declaration potentially in a different module entirely. Notice that in the figure, for these aspects, we choose different names for the children in the production's signature. (Here we use `a` and `b` instead of `l` and `r`.) Thanks to this repetition of the signature, renaming of the names of children of any production does not have any non-local effect on attribute grammar specifications.

3.1.5 Remote/collection attributes

Remote and collection attributes [59] were introduced to allow information to flow in non-local ways, for instance globally, across a subtree, or backwards from reference attributes to the referenced node. (That is, a variable use-node may have a reference to its definition-node, and one way of providing a definition-node with references to all its uses would be for the use-node to reach through a reference attribute and alter a collection attribute on the definition-node.) Our system is primarily concerned with extensibility, and we're not sure how to reason modularly in systems with "remote effects" like this. As a result, our system does not support remote/collection attributes in this sense, but we do have a *localized* notion of *collection attributes*.

Instead of reaching across references to change values, we allow locals within a production to open themselves up to modification by other aspects of the same production. Thus, information still only flows locally and references allow us only to observe, not change, information from elsewhere in the tree. This feature permits a host language to open up custom extension points that can be made use of by language extensions. We note this as a feature of Silver, but we will not make any significant use of it in this thesis.

3.1.6 Well-definedness

Attribute grammars come with a notion of *well-definedness*, and each of the extensions above extend this notion accordingly. Well-definedness can be broken into two parts: *completeness* and *non-circularity*. Non-circularity is simply the assertion that no attribute *instance* (that is, the "box" on a particular node in a decorated tree within which we will place that attribute's value) will, during the course of evaluating its equation, ever come to depend upon itself. Non-circularity is often dropped from the definition of well-defined, however, for two reasons. First, some attribute grammar systems incorporate circular attributes [60], which treat circularity as least fixed point iteration, and is thus not considered inherently ill-defined. Second, even without circular attributes, if the language is lazily evaluated (as Silver is) or otherwise has non-strict constructors of data, then it is possible to construct "infinite data structures" or streams, and thus is also not inherently ill-defined. (Each of these are still capable of producing nontermination, however traditional attribute grammar circularity is decidable, while these are not. As a result, the circularity issue is simply dropped from what makes the attribute grammar "well-defined" and swept under the rug of potential nontermination in evaluating any expression in a general purpose language.)

The last part of well-definedness, completeness, is traditionally defined as a whole-program analysis that simply ensures every attribute has a defining equation on every relevant production. This includes synthesized attributes within the body of each production, but also inherited attributes given to each child. This analysis is whole-program simply because we need to know about every attribute on every nonterminal in order to analyze each production and its equations.

3.2 Silver

Silver [61] is our attribute grammar-based programming language that supported (prior to this work) the above described features of attribute grammars, and a few more minors ones. The central design consideration for Silver is support for composable language extensions. To that end, some enhancements to attribute grammars have been deliberately excluded because we do not know how to reason about their extensibility properties (for example, circular attributes and remote collections, both mentioned above.) This thesis is primarily concerned with extending Silver with new features, developing a framework for reasoning about Silver programs (in particular, language extensions) and their composition, and demonstrating those capabilities.

Silver has been previously used to develop composable language extensions [62, 63, 64, 65]. However these extensions were "unverified": like many of the other surveyed systems, there was no capability to ensure these extensions did, in fact, compose gracefully. The extent of the demonstration was that several example extensions could be built, and these were then composed together automatically without any evident problems. To accomplish this, in Silver's case, the extensions made use of forwarding in what was believed to be a safely composable fashion.

The central contributions of this thesis involve extending this story for Silver. In particular, we will develop an analysis (with an associated set of restrictions on extensions) that is capable of guaranteeing the composability of extensions written in this manner. Further, we will study the possible ways that extensions written in Silver can interfere with each other, and find a way to prevent this from producing unexpected behavior.

As a general approach to compiler implementation, Silver is designed to be quite

appropriate for implementing compiler front-ends. That is, it is well suited to parsing (via Copper, see below), analyzing the abstract syntax tree, and transforming trees. Its current design is not well-suited to non-tree analysis activities (such as iterative algorithms over graphs), however. This means our target compiler design will likely go from source code to an intermediate language and then hand off to another system to complete compilation (lower level optimizations, code generation, etc). This could be a compiler back-end (such as LLVM) or it could simply be pretty printing a program (that has had all extensions translated away) to hand off to a traditional compiler. Thus, our target domain of extensibility is the concrete and abstract syntax of the language. When we speak of domain-specific optimizations, we expect these to be high-level optimizations applied to the AST, rather than lower level optimizations concerned with basic blocks and instructions.

3.3 Copper

This thesis is not concerned directly with concrete syntax, thanks to an "off the shelf" solution provided by Copper [66, 22]. Copper is a context-aware scanner and parser generator that is ideal for language extension for several reasons. The context-sensitivity of its lexer allows the introduction of keywords in language extensions without causing clashes between extensions, as the parsing context is sufficient to disambiguate them. This is ensured by a *modular determinism analysis* [67, 68] that, although it makes certain restrictions on the syntax that can be introduced in an extension, confines the possible errors at composition time to conflicts between "marking terminals," which will be explained shortly. These conflicts are the expected kind that programmers can easily handle, as they routinely do essentially the

same thing for ordinary libraries (renaming or prefixing clashing symbol names, for example, when two imported libraries export the same name.)

The modular determinism analysis works roughly as follows:

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

Here, we ensure that an extension passes the analysis ($\mathit{isComposable}(CFG^H, CFG_i^E)$), and that the result of composing the host language and this single extension does not have any conflicts ($\mathit{conflictFree}(CFG^H \cup \{CFG_i^E)\})$). As a result, we can compose *any number* of such extensions and the result will also have no conflicts, and we can therefore build an LR parser. This ability to achieve a global property by analyzing individual artifacts in isolation is why we call this analysis *modular*.

We apply an additional check to each extension ($\mathit{isComposable}(CFG^H, CFG_i^E)$), about which we will not go into in exhaustive technical detail. But roughly speaking, it imposes two restrictions at the *boundaries* between host and extension syntax. First, we call a transition from host language (an existing nonterminal in H) to extension syntax a *bridge production*. All bridge productions must have the form:

$$\mathit{HostNT} \texttt{::=} \ \mathit{MarkingToken} \quad \text{<extension syntax>}$$

The *marking token* signals an unambiguous transition from host syntax to extension, and must not be preceded by anything else in the right-hand side of the production. If two marking tokens clash, then Copper allows the user to give them a *transparent prefix* that simply precedes the token, disambiguating them. The prefix is essentially a module indicator (like `std::cout` in C++ or `os.path` in Python.)

The second restriction concerns transition from extension syntax to host language syntax—or in a sense, the transition back again. Given an extension production with

a host nonterminal in its signature, which looks something like:

$$\mathit{ExtNT}\texttt{::=}\ \ldots\quad \mathit{HostNT}\quad T\quad \ldots$$

We require the terminal T already be in the *follow set* of *HostNT* in the host language. In other words, there must already be syntax in the host language where a *HostNT* is potentially followed by a T. In real world languages, this might mean an expression could easily be followed by a close parenthesis `')'` but a statement might not be permitted to be followed by anything but a `';'`. This is to signal an unambiguous transition back from the host language to the extension without involving new terminals or rules added to the host parsing table which may cause conflicts.

For this thesis, we consider the syntactic composition problem solved by Copper, though there is undoubtedly room for improvement. Instead, we focus on providing the equivalent of the modular determinism analysis, but for composing semantics (attribute grammars), instead of composing syntax (LR grammars).

Chapter 4

Integrating AGs and functional programming

This chapter will present a subset of Silver which we call AG. Our aim with AG is to distill Silver down to its essential features for two purposes. First, we wish to examine those particular issues involved in blending the features of an attribute-grammar based language with those of modern functional programming languages (the subject of this chapter.) Second, it will serve as the language on which we develop an analysis related to ensuring well-defined composition in chapter 5. And so, AG contains all features of Silver that are interesting for these purposes.

Attribute grammars play a central role in our story of how language extension can be accomplished. But this means host language compilers must be implemented in an attribute grammar-based language. As a result, we have two major goals for this language.

First, it must be analyzable for the purposes of reasoning about extensibility and composition. Attribute grammars were originally described in an expression-language agnostic way. Some attribute grammar-based languages provide this expression language by essentially hybridizing with another language (such as Java or Haskell) which makes this sort of analysis difficult. If attribute equations are simply arbitrary Java code, then we may have some trouble understanding how they might interact when composed with an extension. For example, that code might involve mutation, and extensions can easily cause an unexpected mutation to happen multiple times or

zero times. As a result, AG is a programming language in its own right, in order to limit the power of expressions to that which we can reason about in a modular way.

But equally important, if we are to insist on host language compilers (and extensions) being written in this language, it must be a fundamentally good one. For example, some other kinds of attribute grammar-based languages chose an extremely limited expression language, leaving the programmers quite constrained, especially compared to general purpose languages they are used to. This often makes programmers really feel like they are trapped in an inferior sort of *specification* language, instead of the *programming* language they would prefer to use to implement compilers. And so, in this chapter we reach for functional programming as a means to make this language less cumbersome and more amenable to the modern development of large programs. In particular, we develop AG as, in essence, a (nearly) general-purpose purely-functional programming language, but with attribute grammars as their fundamental data abstraction mechanism (instead of algebraic datatypes, or objects.)

AG is a language with the attribute grammar features discussed in chapter 3, but with several features typical of functional languages as well. This includes the integration of a Hindley-Milner-style type system, parameterized nonterminals, an attribute grammar equivalent of generalized algebraic datatypes, functions, pattern matching on attribute grammars, and some useful ways of leveraging types that are specific to attribute grammars. We also identify and explore some difficulties in pulling off this integration, and point to a few areas in the design space that might be interesting future work.

This chapter is based on our previously published paper "Integrating functional programming and attribute grammar language features"[69]. We will begin in sec-

tion 4.1 by describing in more detail our goals for the language design. Following that in section 4.2, we will introduce the syntax and semantics of AG, including some discussion of interesting design points and alternatives. In section 4.3, we will describe our type system for AG, again considering some design alternatives, while highlighting points of friction and synergy between the AG and functional styles. Useful integration of pattern matching without compromising the extensibility properties, and without unexpected behavior, requires a number of careful design decisions, considered throughout section 4.4. Finally, we conclude with some consideration of the language's extensibility properties (section 4.5) and related work (section 4.6).

4.1 Language Design Goals

Attribute grammars come with a straightforward means of composing together fragments (or modules.) Because each module consists only of a collection of unordered declarations, we can take composition of several modules to be the simple union of the set of declarations from each module. Preservation of this easy composition property is the first goal for the design of an implementation language for language extensions.

The *forwarding* feature was invented to allow introducing new productions without causing problems with other extensions that introduce new attributes. While mere composition of fragments is the first hurdle, the next battle is to ensure that the resulting composed attribute grammar is well-defined. Each extended language (e.g. $H \lhd E_1$) may be well-defined, but composition leaves us no assurances about the result: $H \lhd (E_1 \uplus_{\emptyset} E_2)$ (this assurance is the subject of the next chapter.) Forwarding is intended to make it *possible* for the result to still be well-defined, as equations for new attributes on new productions can be evaluated via the forwarded-to tree.

And so the second goal for our language is to make sure we do not compromise this extensibility: we should still be able to introduce both syntax and analysis in extensions, with the obvious conflict resolved using forwarding.

To make use of the properties of attribute grammars, the host language and extensions must all be implemented in this special language. Since these are large programming tasks, the language must be up to the challenge of modern development. To that end, we seek a way to integrate a variety of features from modern functional programming languages into this attribute grammar-based programming language. But above all, we must do so while preserving the composability and extensibility of the language.

Finally, the language should be such that it is possible to reason about the program, extensions, the composition process, and the resulting artifact and its behavior. Many types of "extensible" systems have an unfortunate tendency to support plug-ins by simply exposing internal mutable state and special event hooks. While this solves the problem of not being able to write a certain kind of extension (just mutate the program state however needed), it rather thoroughly precludes any hope of reasoning about the behavior of a system, especially under composition with other unknown extensions. Because they may do anything, we know nothing.

4.2 The Ag Language

A number of features from the full Silver language are omitted from the language Ag, as we wish to focus on those parts of the language that are interesting from a typing or semantics perspective and are generally applicable to other attribute grammar-based languages. To that end, terminals (and other components related to

parsing and concrete syntax), primitive types, collection attributes, and many forms of syntactic sugar are omitted from AG. There are no major difficulties in extending the contributions of this thesis to the full language (except for collection attributes in later chapters, but we hope to remove them from future versions of the language.) Silver is a proper superset of AG, and examples of Silver programs were shown in the previous chapter.

Figure 4.1 describes the grammar of the language AG. Variables (e.g. productions, local variables, and trees) are denoted x (for values that must be the name of a production, x_p), and we follow the convention of denoting nonterminal names as n, attribute names as a, and type variables as v. Sequences of syntactic elements are denoted using overbar notation; for example, $\overline{v}$ denotes zero or more type variables. In this figure, the major pieces of syntax you see are top-level declarations (D), equations within productions (Q), and expressions (E).

A program in AG is a series of declarations, denoted D. These declarations would normally be mutually recursive, but for simplicity of presentation, we consider them in sequence in AG. (This choice would have semantic implications: type inference on mutually recursive definitions is not able to do generalization the same way that it can on sequential definitions. However, note that the language AG also requires explicit type signatures for all top-level declarations, and so the inference problems of mutually recursive definitions at that level are actually avoided, not glossed over, by these choices.) Most forms of declaration should be relatively standard for attribute grammars. We take the view of attributes being declared separately from the nonterminals on which they occur, which is a common (but not universal) design choice. Likewise, we group attribute equations in association with their productions, rather than each equation as a separate top-level declaration.

variable naming convention

n	nonterminal names	a	attribute names
v	type variable names	x	ordinary variable names
		(x_p	production names)

types

T ::= v | $(T_r$ `::=` $\overline{T_a})$ | n `<` $\overline{T}$ `>` | `Decorated` n `<` $\overline{T}$ `>`

declaration series

D ::= $\cdot$
| `nonterminal` n `<` $\overline{v}$ `>` `;` D
| `synthesized attribute` a `<` $\overline{v}$ `>` `::` T `;` D
| `inherited attribute` a `<` $\overline{v}$ `>` `::` T `;` D
| `attribute` a `<` $\overline{T}$ `>` `occurs on` n `<` $\overline{v}$ `>` `;` D
| `production` x_p x_l `::` n `<` $\overline{T}$ `>` `::=` $\overline{x_r :: T}$ `{` $\overline{Q}$ `}` D
| `aspect` x_p x_l `::` n `<` $\overline{T}$ `>` `::=` $\overline{x_r :: T}$ `{` $\overline{Q}$ `}` D
| `default` x_l `::` n `<` $\overline{v}$ `>` `::=` `{` $\overline{Q}$ `}` D

production equation series

Q ::= $\cdot$
| x `.` a `=` E `;` Q
| `forwards to` E `{` $\overline{A}$ `};` Q
| `local` x `::` T `=` E `;` Q

expressions

E ::= x | $E(\overline{E})$ | `\` $\overline{x :: T}$ `->` E
| E `.` a | `decorate` E `with {` $\overline{A}$ `}` | `new` E
| `case` E `of` $\overline{P}$ `->` $\overline{E_p}$

patterns

P ::= $x_p(\overline{x})$ | `_`

inherited decoration equations

A ::= a `=` E

Figure 4.1: The language AG

We extend nonterminal declarations to be parameterized by a series of type variables ($\overline{v}$) in angle brackets. We will adopt the convention of omitting the angle brackets whenever this list is empty. Attribute declarations, too, are parameterized by a series of type variables, which are then considered bound and may be referenced within the attribute's type. The `occurs on` declaration names a nonterminal us-

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

Figure 4.2: Parameterized declarations: Pairs and ASTs.

ing only type variables, but then names an attribute supplying concrete types. The nonterminal's variables are considered bound for these types. For example, in figure 4.2, we see `snd<d>` where `d` is a concrete type because it is bound by `Pair<c d>`. In the second example, we see a parameterized attribute (`ast`) bound with a regular concrete type (`AbstractExpr`), directly on another unparameterized nonterminal (`ConcreteExpr`.) We will go into considerably more detail about these sorts of issues when we discuss the type system in section 4.3.

Production declarations consist of a name (x_p), a signature that determines both the nonterminal this production constructs (n) and the parameters it takes ($\overline{T}$), and a production body that consists of a series of equations (Q). The name x_l given to the left-hand side of the signature refers to the node itself, and is used principally to define synthesized equations and to access the inherited attributes given to that node. Each of the children (on the right-hand side of the signature) is also given a type and a name (x_r `::` T) which is used to access synthesized attributes from child nodes, and to define equations giving inherited attributes to each child. This is explained with an example in figure 4.3. Finally, notice that the nonterminal the production constructs (n) is parameterized by types ($\overline{T}$) and is not restricted to only *type variables*. That is, a production signature can be specific to a type (`Nt<Boolean> ::= ...`) and is not required to be general over all types (`Nt<a> ::= ...`). This is the syntactic difference allowing the equivalent of generalized algebraic data types

`production root`	Declaring the production `root`.
`r::Root ::= e::RepMin`	Constructs nonterminal `Root` (case-sensitive), and
`{`	takes child argument of nonterminal type `RepMin`.
`  r.rep = e.rep;`	Synthesized equation defining `rep`
	using child's synthesized value `rep`.
`  e.global = e.min;`	Inherited equation defining `global` on the child
`}`	using child's synthesized value `min`

Figure 4.3: A production from figure 3.3, with its components explained in terms of Ag.

(GADTs, discussed later in section 4.3.1.)

We provide two other ways of writing equations for productions, besides in the body of the original production declaration. The first are *aspects* (described in section 3.1.4 that appear syntactically to be almost identical to production declarations. The difference is that aspects name an existing production, rather than declaring a new one. The signature of the production is repeated to make the code easier to understand in isolation and to make name changes a purely local change, and so the aspect can choose its own names for the LHS and RHS values. The purpose of an aspect is to allow equations to be written for a production separately from the module that declares the production.

The second unusual top-level declaration is a way of providing *default* equations for the productions of a nonterminal. This syntax mimics an aspect except that there is no named production, no right-hand side (RHS) signature, and the left-hand side (LHS) nonterminal's parameters are restricted to type variables only. These declarations only allow synthesized equations to be defined on the LHS, and apply to all *non-forwarding* productions that do not have an explicit equation. There are several reasons why defaults apply only to non-forwarding productions, but the most basic reason is that forwarding productions already have a better way to compute a

value for that attribute[1]. Default productions appear to be syntactic sugar (there is nothing they do that we could not do in some other way), however they are useful later in defining the semantics of pattern matching (and they merit attention in later chapters), and so we include them in AG.

Equations (Q) come in several forms. Attribute equations may define the values of synthesized attributes for the LHS, or inherited attributes for the RHS, depending on what x refers to. Thus, several forms of attribute equation have just one syntactic form. Local equations are part of our support for higher-order attributes, and permit new values to be bound and decorated within the production's body. Attribute equations may be written defining inherited attributes for locals, as well.

Forward equations that indicate the tree a production forwards to are just another form of equation that can be written as part of the production body. The semantics of forwarding are described in the previous chapter. Here we take an expression E that should evaluate to a tree that is intended to be *semantically equivalent* to a tree rooted by this production. The set of inherited equations ($\overline{A}$) permit the inherited attributes given to this tree to differ from what are given to the original tree, which would otherwise be copied over unmodified. We often do not wish to change any of these inherited attributes, and so omit this block entirely (writing just `forwards to` E`;`.)

Expressions are denoted E. Application (tree construction) and abstraction are standard for a lambda calculus, and attribute access is standard for attribute grammars. The `new` and `decorate` expressions will be explained momentarily. And the

[1]This is also the correct behavior for wildcards in patterns, for reasons discussed later on in section 4.4. And it's more easily reasoned about in the analysis of chapter 5. And for defaults to apply to forwarding productions is inherently "interfering" in the sense introduced in chapter 6. Just to foreshadow a few more justifications for this behavior of default equations.

pattern matching expression will be discussed later in section 4.4.

4.2.1 Decorated and undecorated trees

In the grammar of types, T, we have two distinct, but related, types that correspond to every nonterminal n_n (omitting type parameters for the moment):

- The undecorated type, denoted n, is simply a term in the object language, a syntax tree for the grammar. These lack any attributes, and are quite similar to algebraic data types in ML or Haskell.
- The decorated type, denoted `Decorated` n, for trees decorated with attributes, created by supplying an undecorated tree with its inherited attributes, and on which we can access synthesized attributes.

To the best of our knowledge, this type distinction is not deliberately exposed in any previous attribute grammar-based system, and so it merits some attention. The observed distinction between these two kinds of trees goes back at least as far as [70], and a type distinction between decorated and undecorated trees shows up naturally in functional embeddings of attribute grammars (such as in [55, 71]), but with less straightforward notation. But normally, decorated types are hidden behind the scenes (if they are a separate type at all), where they are only implicitly used when accessing attributes from children (or locals.)

To go along with these types, AG includes two expressions to convert between the two: `new` creates an undecorated tree from a decorated tree, and `decorate` creates a new decorated tree by supplying it with a list of inherited attribute equations, denoted $\overline{A}$. These operation are illustrated visually in figure 4.4. (Type rules showing this behavior will follow in section 4.3.)

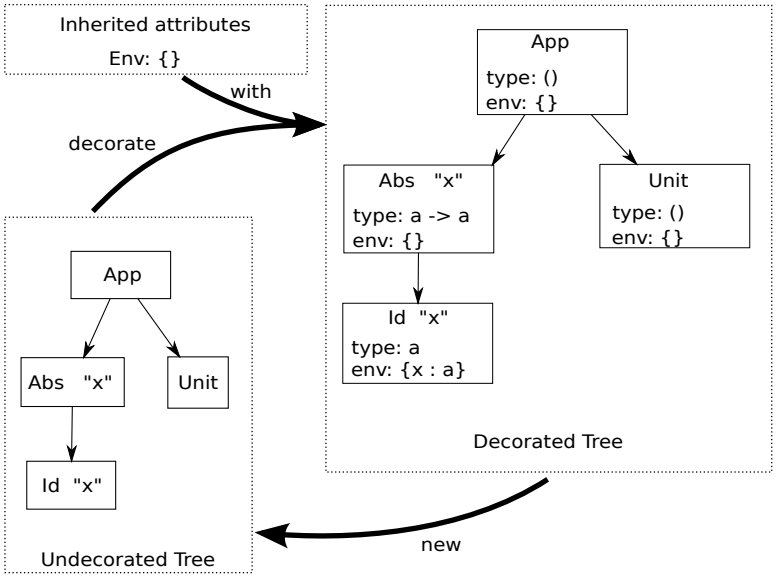


Figure 4.4: Undecorated and decorated trees, and the operations `decorate` and `new` on them. The trees represent the lambda calculus expression $(\lambda x.x)()$.

By exposing the different types, we eliminate the need for different kinds of *attributes*. Extensions to attribute grammars we have previously introduced include higher-order *attributes* and reference *attributes*. These are often treated differently from ordinary attributes. Instead, AG has only ordinary (synthesized or inherited) attributes, but in addition to other primitive types, we can given them undecorated or decorated *types*, which correspond to higher-order and reference attributes, respectively. A higher-order attribute is an undecorated tree, which we can later anchor (and then decorate) using locals, while a reference attribute is an already decorated tree, which we merely pass around as a reference.

This simplifies the language considerably. It also makes using the language slightly safer. AG-based languages like JastAdd [40] and Kiama [9] support both types of attributes, but do so by using the *same* type for each kind of tree. As a result, a

```
synthesized attribute eval :: Boolean;
synthesized attribute neg :: Expr;

production iff
e::Expr ::= l::Expr r::Expr
{ e.eval = l.eval == r.eval;
  forwards to and(implies(l, r),
                  implies(r, l));
}
```

```
nonterminal Pair<a b>
    with fst<a>, snd<b>;

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

Figure 4.5: Left: an excerpt from figure 3.5. Right: a production for the Pair nonterminal of figure 4.2.

decorated tree could at any time be supplied where an undecorated tree was intended, and as a result, not be given the correct inherited attribute values, causing program misbehavior. We have found that directly exposing both types is helpful in quite a large number of ways, and the only cost appears to be a slightly more difficult initial learning curve.

A mini-language for boolean propositions was given in last chapter in figure 3.5. We have reproduced a small piece of it on the left in figure 4.5, to point out a few features of it. The `eval` attribute is an ordinary synthesized attribute (with a primitive `Boolean` type elided from AG). The `negation` attribute was previously described as a higher-order attribute. In AG, we can see this is just an ordinary attribute with an (undecorated) nonterminal type. The `iff` production shows the use of forwarding. It also comes with an equation for one (but not all) synthesized attributes.

Notice, finally, that we see the children of this production used in two different ways. First, we access `l.eval` from the *decorated* tree `l`, and then in the forwarding equation we supply the *undecorated* tree `l` when constructing the forwarded-to tree. This is the learning curve issue we previously mentioned with regards to having

explicit and separate decorated and undecorated types. Productions take undecorated children as arguments, and then implicitly decorate them (since we write inherited equations in the production body), but as this example shows, we often wish to implicitly refer to the original undecorated tree, too. Later on in section 4.3.3, we show how we can use types to accomplish this disambiguation, and thus do not need to write `new(l)` when we want the undecorated child.

The example on the right in figure 4.5 shows the use of the polymorphic production declaration for the simple example of the `Pair` data structure from figure 4.2. Because the types of these children are not syntactically nonterminal types (the type variables may later be instantiated with nonterminal types, but this does not matter,) the production does not implicitly decorate them. Thus, there is no ambiguity when accessing these children, they are always treated as-is. Finally, we note one more convenient piece of syntactic sugar: we can merge occurs declarations with nonterminal declarations using `with`. This allows us to avoid quite a lot of repetition, and works together nicely with the type variable binding semantics of occurs declarations.

4.2.2 Semantics

Prior work gives us a good starting point in defining the semantics of this language. We can re-use an essentially off-the-shelf functional semantics for (higher-order) attribute grammars [72]. Many of the extensions to attribute grammars we have incorporated (reference attributes [58], and forwarding [21]) give a *whole-program* translation back to ordinary, unextended attribute grammars.

We emphasize *whole-program* because these extensions, while they can be translated away when we work with a whole-program, do still give us new capabilities when we have broken our program up into modules, and thus we include them in Ag.

That is, there is still a good reason for keeping them around; they are not merely syntactic sugar. Forwarding, for example, cannot be translated away until we know all attributes that occur on a nonterminal. Our whole purpose in introducing forwarding is that we wish to write modules that do not break each other because they are unaware of some attributes that another module introduces to a nonterminal. But when it comes time to actually execute a program, we are always dealing with a whole program.

The introduction of aspects and default equations poses no special difficulty from a whole-program point of view. Default equations can simply be copied into each (non-forwarding) production that lacks one. Aspects can simply be merged with their original production declarations. (We have aspects because these allow us to write equations in different modules, but when taking a whole-program perspective we can pretend all modules have merged into one.) The only special behavior to note in this transformation is the renaming of variables to make signatures match and to avoid name collisions for locals between aspects. But these are essentially standard program transformation issues.

Each of these translations can be applied without any unexpected interaction with each other. The only major features of AG that taken care of by prior semantics work or the above simple transformations are: the type parameters of nonterminals, and the semantics of pattern matching. We will defer all discussion of pattern matching until section 4.4.

And so, for the remainder of this section, we wish to turn our attention to whether any special problems are introduced in the translation of higher-order attribute grammars to a functional language. In particular, because we have enriched the types on the attribute grammar side, we want to know whether we make any new demands on

the type system of the target functional language. For example, do we need the ability to nest quantifiers within types (like System F), or is the usual type schema stratification sufficient? We answer below that it is sufficient, but perhaps useful extensions could be permitted (beyond AG) if we allowed additional quantifiers. However, our formulation of AG also requires the target functional language of this semantics to support existential types (of a limited sort.)

A typical [72] translation to a typed non-strict lambda calculus will correspond productions with a *semantic function* with roughly the following type:

$$\alpha_N : \text{tuple type of all synthesized attributes on } N$$
$$\beta_N : \text{tuple type of all inherited attributes on } N$$
$$p : \text{a production of AG type } (N ::= \overline{T})$$
$$sem_p : (\beta_N, \overline{\alpha_T}) \to (\alpha_N, \overline{\beta_T})$$

That is, the semantic function corresponding to a production (sem_p) takes as input the inherited attributes given to a node and the synthesized attributes produced by that node's children. It then outputs that node's synthesized attribute and the inherited attributes to give to each child. This tuple must be a non-strict one, or this runs into trouble: almost every interesting attribute grammar would simply be an infinite loop. This function's input includes the output of the children, but its output is the inputs for the children.

These semantic functions can then be *knit*, giving a production type similar to: (N.B. We gloss over the details of this knit function's implementation since they aren't too relevant to us, but it can be found in [72].)

$$prod_p : \overline{T} \to (\beta_N \to \alpha_N)$$

That is, a production takes its arguments and gives back the translated nonterminal type. And that nonterminal type is now also a function, from its inherited attributes to synthesized attributes.

So now we consider the question: how does this story change as a result of introducing type parameters?

We have already mentioned one design choice that considerably simplifies our task here. Whenever a type parameter appears in a production, it is treated as an ordinary value, regardless of whether it ultimately turns out to be instantiated as a nonterminal type. That is, the production cannot have equations that supply attributes to a child of abstract type (since it may not even turn out to be nonterminal type!) This check will appear explicitly in the type rules for productions in the next section. But as a result of this uniform treatment of parameters as values, there are no extra complications with the existing semantics incorporating type parameters on nonterminals, all that happens is quantifiers show up.

$$\begin{aligned}&\text{Given: } p : \forall \overline{v}\ \ (N\langle \overline{T_n} \rangle ::= \overline{T})\\ &\quad \text{Let } \overline{v_n} = fv(\overline{T_n}) \text{ (i.e. the free variables)}\\ &\quad \text{and } \overline{v_p} = \overline{v} \setminus \overline{v_n}\end{aligned}$$

We use v_n to denote those variables in a production that are universally quantified over. We take v_p to be those variables that appear only on the right-hand side of the production signature. These latter variables are universally quantified but in the negative position of a function type, and as a result essentially act as extensionally quantified variables. We then get:

$$\begin{aligned}sem_p &: \forall \overline{v}.\ \ (\beta_N, \overline{\alpha_T}) \to (\alpha_N, \overline{\beta_T})\\ prod_p &: \forall \overline{v_n}.\ \ (\forall \overline{v_p}.\ \ \overline{T}) \to (\beta_N \to \alpha_N)\end{aligned}$$

It is interesting to note there are other options for where type quantifiers could appear here. One possibility (that we may implement in the future) is to permit quantifiers to appear within the types of attributes individually. That is, the elements of the tuple types α or β could be polytypes. We currently forbid this (and this rule will show up explicitly in the type system in the next section) and require attributes have monotypes. This extension would not cause any special problems for the type system (i.e. its still a a system of equalities on monotypes), however we have avoided it so far because we believe it should come with additional syntax (to explicitly indicate the quantified variables in the attribute declaration).

A more invasive possibility would be to permit quantifiers at "decoration time." That is, to permit the translated nonterminal type (which is currently a flat $\beta_N \to \alpha_N$) to have a quantifier as well, this inserting an additional quantifier in the middle of production types. This would permit parameterized attributes to have types related to each other, without involving any type parameters on the nonterminal. For instance, in figure 4.6 we show an erroneous attempt to represent `Bool` values, and implement conditionals as inherited and synthesized attributes. The constraint that all three attributes should have the same type cannot be expressed without adding a type parameter to `Bool`. Doing so means our `Bool` value can only be involved in conditionals of a single type!

The drawback of this extension is that we would have a more difficult (and verbose) time indicating how these references should work. Unlike nonterminals, where we declare the full set of type parameters, the set of parameters for decoration could be extended by any module that introduces a new related set of attributes. Dealing with an open set of nominally addressed type parameters was far too complex of an

```
inherited attribute else<a> :: a;                    production true
inherited attribute then<a> :: a;                    b::Bool ::=
synthesized attribute if<a> :: a;                    { b.if = b.then; }

nonterminal Bool with if<a>, then<a>, else<a>;
-- Error: 'a' is not bound!
```

Figure 4.6: An erroneous program attempting to relate type parameters of attributes without type parameters on the nonterminal.

extension for our tastes. Furthermore, we have yet to find a situation that is probably not better addressed by our first suggested extension (of permitting quantifiers in attribute types.) In the case of `Bool`, for example, we can solve the problem with a single synthesized attribute `cond :: ∀a. a -> a -> a` (taking the then and else branches as parameters to the resulting function instead of as other inherited attributes).

4.3 The Type System

Each of the nonterminals in the language Ag has a typing relation, and many of them differ slightly, and so we will describe them each in some detail. A program in Ag is type correct if there exists a derivation using these rules starting with an empty environment. Algorithmically, we are typically given the term and environment, and we may be tasked, as in the case of expressions, with inferring a type as the output of type checking the subtree. To actually perform inference, we may also have unification variables appear within types in the environment, and these are perhaps best thought of as mutable state that gets updated as unification proceeds.

There is a typing judgment for each syntactic class[2] in the Ag grammar. We will examine each in turn.

[2]Trying to avoid the word "nonterminal" here. The ambiguities of meta-meta-programming!

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

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

$$\frac{fv(T) \setminus \overline{v} = \emptyset \qquad N; P; S; I \cup \{a \texttt{<} \overline{v} \texttt{>} : T\}; O; \Gamma \vdash D}{N; P; S; I; O; \Gamma \vdash \texttt{inherited attribute}\ \ a \texttt{<} \overline{v} \texttt{>::} T \texttt{;}\ \ D} \text{ (D-INH)}$$

$$\frac{\begin{array}{c} fv(\overline{T}) \setminus \overline{v} = \emptyset \qquad a \texttt{<} \overline{v_{a^{env}}} \texttt{>} : T_{a^{env}} \in S \cup I \qquad n \texttt{<} \overline{v_{n^{env}}} \texttt{>} \in N \\ |\overline{v}| = |\overline{v_{n^{env}}}| \qquad |\overline{T}| = |\overline{v_{a^{env}}}| \qquad \alpha(n \texttt{<} \overline{T_\alpha} \texttt{>}) = ([\overline{v} \mapsto T_\alpha] \circ [\overline{v_{a^{env}}} \mapsto \overline{T}])(T_{a^{env}}) \\ N; P; S; I; O \cup \{a@n = \alpha\}; \Gamma \vdash D \end{array}}{N; P; S; I; O; \Gamma \vdash \texttt{attribute}\ \ a \texttt{<} \overline{T} \texttt{>}\ \ \texttt{occurs on}\ \ n \texttt{<} \overline{v} \texttt{>;}\ \ D} \text{ (D-OCC)}$$

$$\frac{\begin{array}{c} n_l \texttt{<} \overline{v_{n^{env}}} \texttt{>} \in N \qquad |\overline{v_{n^{env}}}| = |\overline{T_l}| \\ x_l : n_l \texttt{<} \overline{T_l} \texttt{>}; \overline{x_r : T_r}; \Gamma \cup \{x_l : \texttt{Decorated}\ n_l \texttt{<} \overline{T_l} \texttt{>}\} \cup \overline{x_r : dec(T_r)} \vdash Q \\ T_x = (n_l \texttt{<} \overline{T_l} \texttt{>::=}\ \ \overline{T_r}) \qquad N; P \cup \{x : T_x\}; S; I; O; \Gamma \cup \{x : \forall fv(T_x).T_x\} \vdash D \end{array}}{N; P; S; I; O; \Gamma \vdash \texttt{production}\ \ x\ \ \ x_l \texttt{::} n_l \texttt{<} \overline{T_l} \texttt{>::=}\ \ \overline{x_r \texttt{::} T_r}\ \ \texttt{\{}\ \ Q\ \ \texttt{\}}\ D} \text{ (D-PROD)}$$

$$\frac{\begin{array}{c} n_l \texttt{<} \overline{v_{n^{env}}} \texttt{>} \in N \qquad |\overline{v_{n^{env}}}| = |\overline{T_l}| \qquad x : (n_l \texttt{<} \overline{T_l} \texttt{>::=}\ \ \overline{T_r}) \in P \\ x_l : n_l \texttt{<} \overline{T_l} \texttt{>}; \overline{x_r : T_r}; \Gamma \cup \{x_l : \texttt{Decorated}\ n_l \texttt{<} \overline{T_l} \texttt{>}\} \cup \overline{x_r : dec(T_r)} \vdash Q \\ N; P; S; I; O; \Gamma \vdash D \end{array}}{N; P; S; I; O; \Gamma \vdash \texttt{aspect}\ \ x\ \ \ x_l \texttt{::} n_l \texttt{<} \overline{T_l} \texttt{>::=}\ \ \overline{x_r \texttt{::} T_r}\ \ \texttt{\{}\ \ Q\ \ \texttt{\}}\ D} \text{ (D-ASPECT)}$$

$$\frac{\begin{array}{c} n_l \texttt{<} \overline{v_l} \texttt{>} \in N \\ x_l : n_l \texttt{<} \overline{v_l} \texttt{>}; \cdot; \Gamma \cup \{x_l : \texttt{Decorated}\ n_l \texttt{<} \overline{v_l} \texttt{>}\} \vdash Q \qquad N; P; S; I; O; \Gamma \vdash D \end{array}}{N; P; S; I; O; \Gamma \vdash \texttt{default}\ \ \ x_l \texttt{::} n_l \texttt{<} \overline{v_l} \texttt{>::=}\ \ \texttt{\{}\ \ Q\ \ \texttt{\}}\ D} \text{ (D-DEFAULT)}$$

Figure 4.7: Typing rules for declarations (D) of AG.

$\boxed{N; P; S; I; O; \Gamma \vdash D}$ Declaration sequences

Here N, P, S, I, and O represent sets of, respectively, nonterminals, productions, synthesized attributes, inherited attributes, and occurs-on declarations. Γ represents the normal environment mapping names to types. Elements of N have the shape $n \texttt{<} \overline{v} \texttt{>}$, essentially a name and information about the *number* of type parameters. Elements

of S and I have the shape $a \texttt{<} \overline{v} \texttt{>} : T$, also associating a name with a type (and indicating which type variables are bound within that type.) Elements of P and Γ have the shape $x : \tau$, associating a name with a type. The difference is that for P this is always a production signature, while in Γ this can be any type or type schema. Elements of O, finally, have the shape $n@a = \alpha$, associating the pair of nonterminal and attribute with a type function. This type function accepts a nonterminal type (mostly just for its parameters), and yields the type of the attribute on that nonterminal.

In a typical definition of attribute grammars, it is assumed that we simply have sets representing this information (N, P, and so forth), but this is because the typical presentation of attribute grammars eschews a concrete language in which these declarations exist. For Ag we have a language and we show how to build these sets explicitly. We chose to give D an inductive shape (that is, representing a sequence of declarations, rather than just an individual declaration) in order to show how this information is constructed.

The type rules for D appear in figure 4.7. The rule D-nt for nonterminal declarations is straightforward and adds the nonterminal type to N. We have chosen to simplify the rules somewhat, and simply assume we do not need to worry about redeclarations or name collisions, as this task mostly just adds noise to the rules. The rule D-syn adds the type of the synthesized attribute to S and ensures the type of the attribute is closed under the variables it is parameterized by. The rule for inherited attributes is similar.

The rule D-occ requires some explanation. It's primary task is to create the type function α that computes the attribute's type given a nonterminal's type parameters. For example, the `fst` attribute on `Pair` shown in figure 4.2 would yield a function α such that $\alpha(\texttt{Pair < } S \ T \texttt{ >}) = S$. We write $[\overline{v \mapsto T}]$ to represent a substitution

that maps each type variable to a corresponding type. The definition of α (on the right side of the middle line of the rule D-OCC) looks complex but is quite simple, if we read the composed function from right-to-left. We begin with the type of the attribute looked up in the environment ($T_{a^{env}}$), and rewrite the type variables that attribute is parameterized by ($v_{a^{env}}$) to correspond to those types given in the occurs-on declaration ($\overline{T}$). Next, we rewrite the type variables that appear for the nonterminal in the occurs-on declaration ($\overline{v}$), to become those given by the actual type of the nonterminal ($\overline{T_\alpha}$), i.e. the argument given to α. The rest of this rule merely looks up the nonterminal and attribute, and ensures the number of type parameters match up accordingly.

The rule D-PROD has a few very particular details. The types given for children in a production signature are exactly those types to which the production is applied. Inside the body of the production, however, the left-hand side and all children of nonterminal type should appear with decorated type. The purpose of this is to reflect what the production does: there will be equations inside the production body ($\overline{Q}$) that define inherited attributes for its children. So, if a child is declared as having type `Expr` (as in many of the productions of the boolean language in figure 3.5), then inside the production body, its type should appear to be `Decorated Expr`. To accomplish this, we apply *dec* to the types of the children when adding them to the environment (Γ). *dec* is the identity function, except when applied to a nonterminal type (n `<` $\overline{T}$ `>`) in which case the result becomes the associated decorated type `Decorated` n `<` $\overline{T}$ `>`. We apply *dec* directly to the types as written in the signature T_r, and so type variables are passed on as-is (and even if later instantiated as nonterminal types, they are not decorated, as previously discussed.)

Note that when D-PROD checks the validity of its equations $\overline{Q}$, using the relation

we will discuss momentarily (of the form $L; R; \Gamma \vdash Q$), it supplies R with types unchanged (that is, without *dec* applied.) This is also important, as inherited attributes can only be supplied to undecorated children. Children of already decorated type already have their own inherited attributes, and so we need to know the difference within the production body. Whenever this is important, we can thus choose to look at R instead of Γ.

The rule for aspects is relatively straightforward, it differs from that of productions by not introducing any new value into the environment, and by checking its signature against a previously declared production signature. Likewise, the rule for defaults is simple enough. Like aspects, it introduces no new values into the environment, but unlike aspects it has a signature so restricted that it can apply to every production of a nonterminal. And so, no further check is necessary beyond that the nonterminal exists and we gave it the right number of type variables.

$\boxed{L; R; \Gamma \vdash Q}$ Equation sequences

Since the environment variables N, P, S, I, and O do not change except at the top level of declarations (D), we will omit writing them for the other typing relations and consider them to be implicit. As a result, some of the rules here will mention these without them appearing on the left of the relation. However, we do have interesting pieces of the environment for checking equations: we take note of the signature of the production the equations are for. L is the name and type of the left-hand side of the production signature (the nonterminal the production constructs.) R is the set of names and types for the right-hand side symbols of the production signature (the children.) However, we will also augment R with locals, corresponding to the "anchoring" of higher-order attributes. And so we're generalizing R to really mean

$$\frac{a \in S \qquad (a@n = \alpha) \in O \qquad x : n\texttt{<}\overline{T_n}\texttt{>} = L \qquad \Gamma \vdash E : \alpha(n\texttt{<}\overline{T_n}\texttt{>}) \qquad L; R; \Gamma \vdash Q}{L; R; \Gamma \vdash x\ \texttt{.}\ a\ \texttt{=}\ E\ \texttt{;}\ Q}\ \text{(Q-SYN)}$$

$$\frac{a \in I \qquad (a@n = \alpha) \in O \qquad x : n\texttt{<}\overline{T_n}\texttt{>} \in R \qquad \Gamma \vdash E : \alpha(n\texttt{<}\overline{T_n}\texttt{>}) \qquad L; R; \Gamma \vdash Q}{L; R; \Gamma \vdash x\ \texttt{.}\ a\ \texttt{=}\ E\ \texttt{;}\ Q}\ \text{(Q-INH)}$$

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

$$\frac{\Gamma \vdash E : T \qquad L; R \cup \{x : T\}; \Gamma \cup \{x : dec(T)\} \vdash Q}{L; R; \Gamma \vdash \texttt{local}\ x\ \texttt{::}\ T\ \texttt{=}\ E\ \texttt{;}\ Q}\ \text{(Q-LOCAL)}$$

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

Figure 4.8: Typing rules for production equations (Q) and inherited equations (A) of Ag.

"things we can give inherited attributes to." (The original paper introducing HOAs accomplished anchoring by directly adding them to the signature of productions, marked as not being supplied when constructing the tree, and instead by an equation in the body. Later this was changed to use locals, as we do here, but our type rules reflect how closely related these two approaches are.) This information about the production signature (and locals) is used to distinguish when it is acceptable to defined inherited or synthesized attributes inside the production statements.

The rules Q-SYN and Q-INH are again quite similar, but this time they are not distinguished by syntax. Instead, which rule is used depends on whether an inherited attribute is being defined for a child (or local), or a synthesized attribute is being defined for the left-hand side (i.e. nodes this production corresponds to.) Note that

we use the shorthand $a \in S$ to mean that it is a declared attribute of the appropriate kind, as we no longer care about the type declared for the attribute alone, that will be obtained from the occurs declaration via the function α. The equation for locals is fairly straight-forward, but shows again the same pattern as was seen for children in production declarations: we apply *dec* when placing in the environment of values, but record the type in the production signature unmodified. Note that the rule Q-FWD requires the expression type for the semantically-equivalent tree be undecorated.

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

Here, X is the type of the nonterminal that inherited attributes are being supplied to by `decorate` expressions and `forwards to` statements. The inherited equation rule A-INH is similar to Q-INH except that we obtain the type from the context (X), rather than by looking up a name.

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

This is the standard relation for expressions, except (again) for N, P, S, I, and O (from top-level declarations) as well as L and R (from the production signature) that are implicitly supplied.

The rules E-VAR, E-APP, and E-ABS are slight adaptations of the standard versions of these for the lambda calculus. Notationally, we use ν to represent a fresh type variable, and so in E-VAR, the notation $[\overline{v \mapsto \nu}]T$ performs the usual instantiation of a type schema by substituting fresh variables. The rules E-DEC and E-NEW should be straightforward, based on their descriptions in the previous section, and the visual in figure 4.4. Notice in the rule E-ACC that the expression type is required to be decorated (attributes cannot be accessed from trees that have not yet been

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

$$\frac{\Gamma \cup \{\overline{x : T_x}\} \vdash E : T_r}{\Gamma \vdash \texttt{\textbackslash}\ \overline{x\ \texttt{::}\ T_x}\ \ \texttt{->}\ \ E : (T_r\ \texttt{::=}\ \ \overline{T_x})}\ (\text{E-ABS})$$

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

$$\frac{\Gamma \vdash E : \texttt{Decorated}\ n\ \texttt{<}\overline{T}\texttt{>} \qquad (a@n = \alpha) \in O}{\Gamma \vdash E\ \texttt{.}\ a : \alpha(n\ \texttt{<}\overline{T}\texttt{>})}\ (\text{E-ACC})$$

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

Figure 4.9: Typing rules for expressions (E) of AG, pattern matching can be found in section 4.4.

decorated with attributes.) Notice also that this rule reports a type that is a result of the function α. In section 4.3.2, we consider the problem with this rule, as written, for type *inference* where we must know the final type of the left hand side in order to report any type at all for the whole expression, due to the function α.

4.3.1 Generalized algebraic data types.

An in-depth description of GADTs can be found in [73]. We will not dive deeply into examples of the utility of GADTs, to keep things focused. However, in figure 4.10, we show two classic examples. On the left, the `eq` production constructs the type `Eq<a a>`. The repetition of the type `a` makes this a GADT. Using this type, it is possible to convert from one type to another. We can use a value of type `Eq<a Integer>` to convert values of type `a` to `Integer` by passing them into `in` and retrieving them

```
nonterminal Eq<a b>
  with in<a>, out<b>;
inherited attribute
  in<a> :: a;
synthesized attribute
  out<a> :: a;

production eq
e::Eq<a a> ::=
{ e.out = e.in;
}
```

```
nonterminal Expr<a> with eval<a>;
synthesized attribute eval<a> :: a;

production ifExpr
x::Expr<a> ::= c::Expr<Boolean>
               t::Expr<a>  e::Expr<a>
{ x.eval = if c.eval then t.eval else e.eval;
}
production trueExpr
x::Expr<Boolean> ::=
{ x.eval = true; }
```

Figure 4.10: Two separate examples of the use of generalized algebraic datatypes. On the left, an equality type with type coercion attributes. On the right, an abstract syntax with a type-safe evaluation attribute.

from `out`.

On the right of this figure, we see another classic example, as applied to abstract syntax. The `ifExpr` actually shows an ordinary constructor, nothing too special about it. It is the `trueExpr` constructor, which specifically constructs `Expr<Boolean>`, instead of accepting any type parameter to `Expr`. As a result, we can statically rule out this constructor when we have a value of, say, `Expr<Integer>`. In the figure, we show a type-safe (and unwrapped) evaluation attribute. For `trueExpr`, we can see this giving a new capability: we would, without GADTs, not be able to directly assign `true` as the return value. Inside `ifExpr`, we actually see another capability: we are able to directly access `c.eval` as a boolean value, without having to unwrap some value and inspect whether we obtained the correct type. An abstract syntax with this sort of construction cannot have type errors at all (unless our meta-language has an unsound type system.)

Readers unfamiliar with GADTs need not fear. Although we take some time to support them here for the purpose of writing programs in Silver, they do not play

any significant role in the central contributions of this thesis.

The language AG (having skipped over pattern matching expressions) supports GADTs effortlessly. The type system presented in this section so far needs no changes at all, whether GADTs are allowed or not. The only difference is actually syntactic. If the type of the nonterminal on the left hand side of `production` declarations permits types ($\overline{T}$) inside the angle brackets (as they do in figure 4.1), GADTs are supported. If instead, these are restricted to (non-repeating) type variables ($\overline{n_v}$), then GADTs are disallowed. All of the complication in supporting GADTs appears to lie in pattern matching, as we will see when we come to it in section 4.4.

4.3.2 Polymorphic attribute access problem

By a Hindley-Milner style type system we mean one with a simple essential feature. The program is analyzed and generates as a result a set of constraints, all of which are exactly one kind: that two types should be equal. This short phrase actually packs a lot of details. It is just *types* that should be equal, not type schemas, for instance. This simplifies the unification problem considerably. We also only consider merely equality between types, without more interesting relations, like subtyping. And finally, that equality is the only constraint type we have, not a mixture of different kinds of constraints.

We encounter one important issue in adapting a Hindley-Milner style type inference system to attributes grammars. Typing the attribute access expression `e.a` immediately raises two problems with the standard inference algorithm:

1. There is no type we can immediately unify `e`'s type with. The constraint we wish to generate is that, whatever `e`'s type (call it n), the attribute `a` occurs on

it. That is, we should find $a@n = \alpha$ within O. But this is not expressible as a type equality constraint.

2. There is no type that we can report as the type of the whole expression, without first knowing `e`'s type. The constraint we wish to generate would need to compute (using α) that type, and we cannot do that during constraint generation, without solving some other constraints first.

These problems occur even in the simplest case of parameterized attributes. For example, we cannot know that `e.fst` means that `e` should be a `Pair`, since that attribute may occur on many types, such as a `Triple`. And without knowing `e`'s final type, we can't do any better than report an unconstrained type variable (for now) as the type of the whole expression, which further exacerbates the problem when chaining attribute accesses (e.g. `e.fst.head`.) Any access of `fst` simply requires knowing what type we're accessing `fst` on.

Fortunately, a sufficient level of type annotation guarantees that the types of subexpressions can be inferred before reporting a type for the attribute access expression. The syntax of Ag requires this minimal level of type annotation already. As a result, our type inference algorithm can interleave constraint generation and constraint solving to successfully deal with this problem. This is possible because Ag requires explicit type annotations for every name introduced into the environment. For attribute grammars, this is not a significant burden, because productions and attributes need type annotations regardless. In Ag, it is a cost only for lambdas and perhaps locals, where we might have otherwise enjoyed inference, but explicit types are required.

Despite requiring some annotations, type inference does still provide a significant

advantage since it infers the "type parameters" to parameterized productions. In a prototype implementation of parametric polymorphism in Silver [74] that did not do inference, one needed to specify (for example) the specific type of an empty list literal. For example, `nil<Boolean>()`. Explicitly providing such type parameters quickly becomes tedious.

For AG we have gone with the above solution, as it does not place any new demands on the target functional language our semantics are defined in terms of. However, we wish to note a probable future development. It is possible to resolve this problem using a more powerful type system and inference engine than Hindley-Milner. OutsideIn(X)[75] describes a type inference system where universal quantification is augmented with constraints from the "constraint domain" X. They then describe a particular domain X that includes both type equality hypotheses and type class constraints. This appears to be sufficient to solve the attribute access problem, though we have not worked out all the details. We can use type class constraints to solve half the problem: `e.fst` can take `e`'s type (even if currently just an unconstrained type variable) and simply require it be within a type class corresponding to `fst`. A clever pattern for using type equalities allows us to define *type functions* the unifier understands. We can introduce a type family for each attribute, such as `Typeof_fst<a>`, and for each attribute occurrence an instance, such as `Typeof_fst(Pair<a b>) = a`. This allows us to avoid having to (externally to the constraint solver) look up a function like α because we can instead encode it directly in the type system (that is, as constraints). As a result, we can report the type of an expression like `e.fst` as `Typeof_fst<`T`>` where T is `e`'s type, even if still unconstrained. These extensions are enough to once again look on type inference as a two phased process (constraint generation and solving.)

4.3.3 Putting types to work

In the rule D-PROD, children of nonterminal type are added to the environment in their decorated form for the body of the production (using the function *dec.*) While this is the correct behavior for attribute grammars, it can occasionally be inconvenient and unexpected. Sometimes we wish to actually refer to the original value as it was given, and this can lead to a tedious proliferation of `new` appearing in those cases.

What we'd like is to have these names refer to *either* of their decorated or undecorated values, and simply disambiguate based upon type needed. The example grammar in figure 4.5 is already relying on this desired behavior, in fact. In the `and` production, we happily access the `eval` attribute from the decorated children, when defining the equation for `eval` on this production. But, we also use `r` and `l` as undecorated values in the forwarding equation. As currently written, the type rules would require us to write `new(s)` in the latter case, because the higher-order attribute expects an undecorated value, and `s` is seen as decorated within the production.

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

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

Unfortunately, simply introducing this rule leads to nondeterminism in the type checking algorithm. With it, there is no obvious way to decide whether to use this rule or E-VAR, which is problematic for inference.

To resolve this issue, we introduce a new pseudo-union type of both the decorated and undecorated versions of a nonterminal. But this type, called *Und* which we pro-

nounce "undecorable" (meaning, a type we can implicitly undecorate,) will also carry with it a type variable that is specialized to the appropriate decorated or undecorated type when it is used in one way or the other. This restriction reflects the fact that we can only choose one of these values or the other. This type can be thought of as a type variable, but with only two possible values. *Und* is brought into play by altering the *dec* function to turn undecorated nonterminal types into undecorable types, rather than decorated types. Concretely, $dec(n\texttt{<}\overline{T}\texttt{>}) = \forall a.\ Und(n\texttt{<}\overline{T}\texttt{>}, a)$. Each use of a variable of this type will get its own fresh internal type variable.

This type is given its special behavior by altering the unification rules. Unification proceeds by finding substitutions for unification variables such that an equivalence relation on terms holds. We can alter this equivalence relation on types ($\mathcal{U}(x, y)$) to have unusual rules for *Und* as follows:

$$\mathcal{U}\big(Und\langle n\texttt{<}\overline{v}\texttt{>}, a\rangle,\quad n\texttt{<}\overline{v}\texttt{>}\big)\ \texttt{:-}\ \mathcal{U}(a, n\texttt{<}\overline{v}\texttt{>})$$

$$\mathcal{U}\big(Und\langle n\texttt{<}\overline{v}\texttt{>}, a\rangle,\quad \texttt{Decorated}\ n\texttt{<}\overline{v}\texttt{>}\big)\ \texttt{:-}\ \mathcal{U}(a, \texttt{Decorated}\ n\texttt{<}\overline{v}\texttt{>})$$

$$\mathcal{U}\big(Und\langle n\texttt{<}\overline{v}\texttt{>}, a\rangle,\quad Und\langle n\texttt{<}\overline{v}\texttt{>}, b\rangle\big)\ \texttt{:-}\ \mathcal{U}(a, b)$$

Written in Prolog-like notation, this essential permits us to consider an *Und* type equivalent to either the decorated or undecorated type, and then commits to that choice thereafter.

Now, suppose we have the following (admittedly contrived) types below, and we attempt to type the following expression:

```
bar :: (Baz ::= Expr)
foo :: (Baz ::= a  (Baz ::= a)  a)
child1 :: Und<Expr b>
child2 :: Und<Expr c>
```

```
foo(child1, bar, child2)
```

When we visit the first argument to `foo`, we will learn the constraint `a = Und<Expr b>`, equating the first argument's type with the function's first parameter type. Upon visiting the second argument, we will find that `a = Expr` which means (putting both these constraints together) that `Und<Expr b> = Expr`, and as a result of our custom rules, that `b = Expr`. Finally, visiting the third argument, we will learn that `Und<Expr Expr> = Und<Expr c>` and so that `c = Expr`, again by the custom rule. As a result, we correctly discover that both child references are for their respective undecorated trees.

The introduction of this undecorable type is something of a special-purpose hack, but the notation gains are worth it, and it's always nice to use the type system to do our work for us.

4.4 Pattern Matching

Until now, we have been omitting pattern matching from our discussion of the semantics and type rules for Ag. To integrate pattern matching, we have to solve significantly harder design problems than for other features. The biggest difficulty comes in preserving our composability and extensibility goals for the language.

Core to our story for supporting language extensions is the use of attribute grammars and forwarding to keep the language extensible. Whenever an extension introduces a new production, it's also capable of providing attribute equations for existing attributes that occur on the corresponding nonterminal. Similarly, whenever an extension introduces a new attribute, it can provide equations for all existing productions. While a pattern matching expression seems analogous to introducing a new

attribute, as soon as there are pattern matching expressions in the program, we encounter a problem. There's no way we can (in another extension) somehow "extend" a pattern matching expression to handle new cases, like we're able to "extend" an existing attribute to give new equations for new productions. Pattern matching puts two of our goals in direct conflict: we want this feature to make it (in some cases, dramatically) easier to write compilers and extensions in Ag, but we can't introduce it as-is without compromising extensibility.

There is a plausible path to a solution, however. We had included *forwarding* in the language in order to solve a similar-sounding extensibility problem. A pattern matching expression cannot be extended with new cases by another module. But neither can an attribute be extended with new equations, if a module is unaware that attribute exists. If we're to make *independent* language extensions compose without further effort, then these independent modules are unaware of attributes introduced by the others. This problem is taken care of by forwarding, as these attributes can be evaluated on the forwarded-to tree instead. If pattern matching were to behave with respect to forwarding just like attributes do, the problem would be solved. Matching on a forwarding production could instead simply proceed by matching on the forwarded-to tree instead.

Giving pattern matching this attribute-like semantics involves a number of changes, however:

- Attributes are evaluated on decorated trees, not on undecorated ones, and so we should match patterns on decorated trees instead of undecorated ones. Indeed, we are forced to make this choice: evaluation of forwarding equations may require the use of inherited attributes.

- Relatedly, the pattern variables will bind to the decorated children of the root node of a decorated tree, just as attribute equations see the production's children as decorated types.
- We have to make some sort of choice regarding the behavior of wildcards, and we have several options very different from the usual semantics (which we will consider shortly.)
- Finally, and most obviously, we will change the semantics for a failure to match, to re-try on the forwarded-to tree.

One major simplification we can make in our approach to the problem is to only consider the behavior of a "primitive" match expression–matching non-nested patterns on a single value, as in Ag. For fully nested patterns on multiple values (such as in the full Silver language), we can make use of an off-the-shelf compilation technique[76] to automatically transform down to just primitive matching. This transformation is responsible for turning nested patterns on multiple values into a tree of primitive match expressions, that discriminate on one single value at a time, and only bind variables, with no further nesting in the pattern.

We had some concerns about this approach early on, but those were quickly found to be baseless. The initial worry was that we might want to make a different choice for the order in which we attempted to "look through" values to what they forward to, and the normal pattern matching compiler would take this choice away from us. However, it turns out that in practice this ordering problem was a complete non-issue, and the usual pattern matching compiler always seemed to work in the expected way. We believe there were two reasons for this. First, the ordering issue exists with normal pattern matching (without AGs): wildcards already create more-specific and less-

specific cases all of which can match, and programmers already order their patterns naturally with that in mind. Second, for reasons we will elaborate upon in chapter 6, patterns are almost always going to be used to match on non-forwarding productions anyway, and so the potential problem is unlikely to manifest[3]. As a result, we can consider only the semantics of primitive patterns in AG.

One standard feature of pattern matching is exhaustiveness checking: making sure it is not possible for the match expression to fail. When matching on algebraic datatypes, this is possible because we are aware of all possible constructors for this type, and so we can ensure they are all present. At present, however, we lack this property even for attributes: although we have forwarding to allow extensions to "play nice" by only introducing forwarding productions, nothing stops them from introducing new non-forwarding productions and thus causing problems. In the next chapter, we will resolve this problem for attributes, and restore our ability to do exhaustiveness checking for patterns as well.

4.4.1 Behavior

In figure 4.11, we show a very simple example of the use of pattern matching in determining equality of type representations. (We again take a few small liberties in borrowing notation from the full Silver language; new in this example is the use of a list type and some notations for it.) The advantages of the interaction between pattern matching and forwarding quickly become apparent in the example (on the right) of a tuple extension to the language of types. The tuple production gets its equality checking behavior "for free" from whatever tree it forwards to, as is normal

[3]To foreshadow this a little, it turns out that it is not safe to rely upon seeing a forwarding production instead of what it forwards to, as this is one way extensions can "interfere" with each other and misbehave.

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

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

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

Figure 4.11: A use of pattern matching.

for forwarding.

But, in the absence of pattern matching looking through forwarding, this is not sufficient: consider checking two tuples (`tuple([`S, T, U`])`) against each other. The first will (in one step) forward to `pair(`S`, tuple([`T`, `U`]))`, and use `pair`'s `eq` equation, and thus pattern match on the other type, expecting a `pair`. But, without the "look-through" behavior, the pattern will not match correctly, as it will see a `tuple` instead of a `pair`.

With the amended behavior, the failure to match against a `tuple` will be treated like a missing equation on a forwarding production. Just as the `eq` equation for `tuple` was gotten from `pair`, so the pattern matching expression will match `pair` after failing to find a case for `tuple`.

Helpfully, this example also gives us guidance in how we should design the behavior of wild cards (and variables) in patterns. The pattern in the `pair` production matches only against `pair`, and otherwise reports type inequality. If the wildcard were to

match eagerly, even though productions might forward, then this would not give us the behavior we want. The wildcard would eagerly swallow up a value rooted in `tuple` before we ever tried to "look through" and see that it forwarded to `pair`. As a result, we are motivated to make wild cards apply *only* to *non-forwarding* productions. This is exactly the behavior we get from default equations in AG, and that is no accident.

The pattern matching compiler [76] (which reduces multiple values with nested patterns as we see in Silver down to primitive matching in AG) works by choosing a value to match on, and handling three cases of what patterns might match on that value: all-constructor, all-variable (including wildcards,) and mixed patterns. The all-constructor case reduces to the primitive match expression, like the one we will describe. Nothing here needs to change about the pattern compiler, since we accomplish our goals by changing the behavior of this primitive match construct. The all-variable case reduces to a let expression. This turns out also to be perfect behavior: we prefer not to look through a forwarding value if there is no need to do so. For the mixed pattern case (containing some constructors, and some variables), the pattern matching compiler reduces this to a primitive match with a specialized "failure case." The primitive match contains only the constructors from the mixed patterns, and the "failure case" is no longer failure: it simply proceeds by let-binding the variables from the mixed case. All this means we have to make no changes at all to the usual pattern matching compiler to get our desired behavior.

4.4.2 Typing pattern matching expressions

The rules for typing pattern matching expressions and pattern match rules appear in figure 4.12. For the expression itself, the only interesting piece to note is that the scrutinee must have decorated type. All the interesting bits are in the rule for

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

$$\frac{\begin{array}{c} x_p : (n\texttt{<}\overline{T_l}\texttt{>}\ \texttt{::=}\ \overline{T_r}) \in P \\ \theta \in mgu(\overline{T_l = T_n}) \qquad \theta(\Gamma, \overline{x_v : dec(T_r)}) \vdash E : \theta(T) \end{array}}{\Gamma \vdash x_p(\overline{x_v})\ \ \texttt{->}\ \ E : n\texttt{<}\overline{T_n}\texttt{>} \to T}\ \text{(P-PROD)}$$

Figure 4.12: Typing rules for pattern matching (E-CASE and P) of AG.

patterns.

$$\boxed{\Gamma \vdash E : T \to T}\ \text{Patterns}$$

The typing judgment for patterns is supplied with both the scrutinee type and also with the return type of the expression. There are a few subtleties in the rule for patterns. Let us start with the less obvious. Note the application of *dec* to T_r in P-PROD. The use of this function appears here, because the children being extracted from the scrutinee are decorated–just as the children in a production appear decorated within its body, to its equations. But also notice this function (*dec*) is applied *prior* to the use of θ. We will momentarily discuss θ in depth, but for the moment we can think of it as being information about how type variables are instantiated for the scrutinee. That is, instead of writing an equation solely about a general `Pair<a b>` we will be specifically writing an expression about a concrete instantiation, for example the scrutinee's type may be `Pair<Expr Boolean>`. Thus, *dec* must look at the type before the instantiation shows up, otherwise we would be under the erroneous impression that the first element of the pair was a `Decorated Expr`.

The explicit use of θ in the type rule P-PROD is the cost that we must pay for supporting GADTs. The approach we show here for handling GADTs in patterns are

adapted from an especially simple to implement approach to handling them[73]. In that paper, much attention is paid to a notion of *wobbly* and *rigid* types. Thanks to the concessions in type annotations we must make due to the attribute access problem discussed in section 4.3.2, all bindings in Ag can be considered rigid in their sense[4], leading to our slightly simpler type rules.

The essential idea is to compute a *most general unifier* (θ) between the pattern scrutinee's type and the result type of the production[5]. The explicit application of θ to the environment and type makes these assumptions visible while checking that match rule's expression, but also means these assumptions are "undone" when we move on to the next match rule. In effect, all this rule is really stating is that whatever type information we learn from successfully matching a particular GADT-like production stays confined to that branch of the pattern matching expression. In this way, when pattern matching on an expression of type `Foo<a>` and matching a production that constructs a `Foo<Integer>`, we type check the corresponding expression under a "world" where `a` no longer exists because it has been rewritten away to `Integer`.

One down side to the introduction of pattern matching (with GADTs supported in this manner,) is that we may be unable to infer the resulting type of a pattern matching expression in a bottom-up way. The "attribute access problem" we mentioned earlier was resolved on the assumption that we would always be able to infer types for subexpressions. Although this represents a hole in that reasoning, it's a small one. It requires an attribute access to occur directly on a pattern matching subexpression that matches exclusively to GADT-like productions, with no interven-

[4]Well, in practice *almost* all are rigid. Where we do have a difference, we choose to instead simply raise a (possibly unexpected) type error, rather than deal with the complication of "wobbly" types.

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

ing explicit result type. That is, an expression like `(case x of eq() -> y).attr`, where our algorithm is unable to discover the result type of the return expression. Normally, we would find the result type of the case expression by seeing how it is used (and then later checking this against the type of the expression `y` inside the match rule,) but in this case we only use it by accessing an attribute–which doesn't help us discover its type due to the attribute access problem. We have yet to have any users run afoul of this small hole in our type inference engine in practice, and so we've chosen to simply live with the potentially unexpected error message.

4.4.3 Semantics

In giving the semantics for pattern matching, we run into a serious technical issue. We had noted in passing earlier that GADTs were quite easy to support on the attribute grammar side of things, but they caused a complication in our typing rules for pattern matching (the need for θ in the rules for patterns). However, there is in fact a duality to this issue, something patterns make easy that attribute grammars make hard. Consider a simple pattern, such as one within a `sum` function matching on a `List<Integer>`. But now consider what this would be equivalent to: a `sum` attribute with type `Integer` that occurs on `List<Integer>`... except that a concrete type like `Integer` is illegal in the parameters of the nonterminal in an `occurs on` declaration! Only variables may appear there, not concrete types.

We chose to permit only variables there (and thus forbid these "partial occurrences") for two reasons. First, it is simple, and prevents a proliferation of hacks (like θ) from polluting all the type rules for the language. Second, this syntactic restriction lets us re-use previous work on the semantics of attribute grammars. Translation down to a functional language for productions of parameterized nonterminals is iden-

tical to simple nonterminals, except that parameterized types are used instead of simple ones. However, these "partial occurrences" throw a wrench into that machinery. What does it mean for an attribute like `sum` to occur only on `List<Integer>`? Does that mean productions have different semantic functions, depending on the type parameters of the nonterminal?

We will sketch a solution to these issues in section 4.4.4 by using a more powerful type system. But first, we proceed to give a semantics based on the restriction that pattern matching expressions are only used on decorated types where the parameters are held abstract (identical to how equations in productions work: we can't know what the types `A` and `B` are within a `pair<A B>` production.) This is a somewhat unreasonable restriction, as it precludes our simple of example of the pattern matching in a `sum` function. However, it is not useless exercise, as the shape of the translation will actually be identical for the general case: all we're really missing is an expressive enough type system in the target language.

In figure 4.13 we give a semantics to pattern matching by translation to attributes, under the assumption that there are only variables in the type parameters of the nonterminal type of the scrutinee. The places where this assumption shows up have been highlighted in the figure. This translation analyzing a case expression and yields a set of declarations (D) that declare a synthesized attribute and its equations that are equivalent to the pattern matching expression. The type of the generated attribute is a function, from the free variables that appear in the match rule's expression, to the result type of the pattern matching expression. After generation of this attribute and these declarations, the pattern matching expression can be replaced (to use the notation from the figure) with $E.\nu(\overline{x_{free}})$. That is, simply accessing the generated attribute (ν) from the scrutinee (E) and applying the free variables it requires.

$$[\![\Gamma \vdash \texttt{case}\ \ E\ \ \texttt{of}\ \ \overline{P\ \ \texttt{->}\ \ E_p} : T]\!] = \begin{array}{l}\texttt{synthesized attribute}\ \ \nu\texttt{<}\overline{v}\texttt{>::(}T\texttt{::=}\ \ \overline{T_{free}}\texttt{)}\texttt{;} \\ \texttt{attribute}\ \ \nu\texttt{<}\overline{v}\texttt{>}\ \ \texttt{occurs on}\ \ n\texttt{<}\ \overline{v}\ \texttt{>}\texttt{;} \\ [\![\Gamma \vdash \overline{P\ \ \texttt{->}\ \ E_p} : n\texttt{<}\ \overline{v}\ \texttt{>} \to T]\!]\ \nu\ \overline{x_{free}\texttt{::}T_{free}}\end{array}$$

where : ν is a fresh name $\quad \overline{x_{free}} = fv(\overline{P\ \ \texttt{->}\ \ E_p})$

$\Gamma \vdash E : \texttt{Decorated}\ n\texttt{<}\ \overline{v}\ \texttt{>} \quad \Gamma \vdash \overline{x_{free} : T_{free}}$

$$[\![\Gamma \vdash x_p(\overline{x_v})\ \ \texttt{->}\ \ E_p : n\texttt{<}\ \overline{v}\ \texttt{>} \to T]\!]\ \nu\ \overline{x_{free}\texttt{::}T_{free}} = \begin{array}{l}\texttt{aspect}\ \ x_p \\ \texttt{top::}n\texttt{<}\overline{T_n}\texttt{>}\ \ \texttt{::=}\ \ \overline{x_v\texttt{::}T_v} \\ \texttt{\{} \\ \texttt{top.}\nu\ \texttt{=}\ \backslash\overline{x_{free}\texttt{::}\theta(T_{free})}\texttt{->}E_p\texttt{;} \\ \texttt{\}}\end{array}$$

where : $x_p : (n\texttt{<}\overline{T_n}\texttt{>}\ \texttt{::=}\ \ \overline{T_v}) \in P$

$\theta = [\overline{v \mapsto T_n}]$

$$[\![\Gamma \vdash \_\ \ \texttt{->}\ \ E_p : n\texttt{<}\ \overline{v}\ \texttt{>} \to T]\!]\ \nu\ \overline{x_{free}\texttt{::}T_{free}} = \begin{array}{l}\texttt{default} \\ \texttt{top::}n\texttt{<}\ \overline{v}\ \texttt{>}\ \texttt{::=} \\ \texttt{\{} \\ \texttt{top.}\nu\ \texttt{=}\ \backslash\overline{x_{free}\texttt{::}T_{free}}\texttt{->}E_p\texttt{;} \\ \texttt{\}}\end{array}$$

Figure 4.13: A translation from fully-typed (thus why the whole judgment appears within brackets) pattern matching expressions to attribute declarations. Highlighted are our restrictions to variables rather than types.

The syntactic restriction to variables only shows up in two places in AG. First is the type parameters of the nonterminal in the occurs-on declaration, and second is the type parameters of the default production. As a consequence, we see that we require the scruntinee E's type to also have only variables for its type parameters.

In the rule for the expression, note that when we write $fv(\overline{P\ \ \texttt{->}\ \ E_p})$, we do not consider the variables bound by P to be free. That is, a match rule like `p(x, y) -> x + y + z` would only have `z` as a free variable.

In the rule for patterns, we see the effect of GADTs in a much more simplified form. When patterns are translated to attributes, we would simply have written different (concrete) types instead of variables for the free variables of the expression. That is, when matching on a `Nt<a>` with a free variable also of type `a`, and we match against a production of type `Nt<Integer>`, we are simply accepting a free variable of type `Integer` now.

And so, with a restriction to type variables, we are able to translate away pattern matching expressions to attributes. This conveniently gives us exactly the same semantics for "missing match rules" as for missing equations. As a result, pattern matching expressions will "look through" forwarding production, and successfully match on trees they eventually forward to.

4.4.4 Towards semantics without the variable restriction

We have previously suggested some extensions we wished to make to Ag over our current formulation. One of these was to permit universal quantifiers in the type of attributes. Another was to upgrade from an ordinary Hindley-Milner style type system to OutsideIn(X), to be able to once again separate the "constraint generation" and "constraint solving" phases of type checking. OutsideIn(X) accomplishes this by permitting type equality constraints to appear in type schemas. These extensions would be enough to solve the variable restriction problem in the translation we've given for pattern matching.

If we allow attributes to have the full form of polymorphic types ($\forall x.C \implies T$ where C is a set of constraints), and we have the more sophisticated type language that can express equality constraints, then the above translation works perfectly with one simple change. The type we given the generated synthesized attribute just needs

the appropriate set of type equality constraints. We can illustrate this easily enough by example. Our earlier example of `sum` on a `List<Integer>` could be expressed as follows (note in particular the type of the attribute):

```
nonterminal List<a>;
synthesized attribute sum<a> :: a ~ Integer => Integer;
attribute sum<a> occurs on List<a>;

production cons
top::List<a> ::= h::a  t::List<a>
{
  top.sum = h + t.sum;
}
```

The syntactic restriction to variables in AG is fine, because we can instead express the equality constraints we need directly. The equations work out fine as well, the expression (`h + t.sum`) is type checked under a context with the assumption that `a = Integer` and so adding the head of the list to an integer is considered type correct. Adopting this type system is not entirely trivial (and we do not explore it in this thesis), as we also need to be able to discharge equality constraints as part of type checking. For instance, to access `sum` on a list, we need to discharge the `a ~ Integer` constraint. This can be because of reflexivity (we're actually accessing it on something we know to be `List<Integer>`) or, as in the above example, it can be because such an assumption already exists in our context (because we are writing an equation for `sum`, it has been introduced as an assumption.)

The semantics are nicely clear though, needing no further change than the upgrade to the underlying type system. What it means, then, for `sum` to occur on a `List<Integer>` is that `sum` always occurs on all `List<a>`, but it would be a type

error to access it unless the constraint that `a` is equal to `Integer` can be satisfied. This alleviates our concerns about "partial occurrences."

Although upgrading the language in this way would lead to a considerable advantage in simplifying the type system and semantics for AG (and making the language considerably more powerful), it also leads to considerable increase in the complexity of type system. We believe that our presentation of the intricacies of integrating attribute grammar and function language features is improved by sticking with a simpler type system, and highlighting those areas where this choice causes significant friction (along with a suggested path to alleviating them.)

4.4.5 Differences between AG and Silver

As we have detailed, pattern matching can be translated to attributes, but that translation comes at the cost of potentially needing many attributes. We have not emphasized this point, because we have focused on the primitive match expression, using a standard pattern matching compiler to reduce down to just these primitive expressions. However, that pattern matching compiler can produce a number of primitive match expressions that is exponential in the depth of patterns. Combined with the fact that we cannot translate some types of patterns to attributes without a stronger type system, this is likely not a good approach for implementing pattern matching, in practice.

Amplifying this problem, the most prolific data structures are probably also those that are pattern matched upon the most. Unless the attribute grammar implementation is specifically designed around mitigating this problem (and Silver is not,) there will be overhead for every attribute. A `List` decorated node, for example, could easily balloon in size with many attributes that are the result of translated-away patterns,

and there are likely to be a very large number of `cons` nodes created by a typical program. The result would not be memory efficient, to say the least.

As a result of all this, we have chosen to directly implement a primitive match expression directly as a part of the core language of Silver, rather than implement it by translation away to attributes. However, we have given its behavior identical semantics to the translation given here.

4.5 Algebraic properties of language extension

In the introduction to this thesis, we described a notation useful for describing the artifacts of language composition, such as H, $H \lhd E_1$, and $L_1 \uplus_g L_2$. This notation was quite abstract, applying to a wide variety of different language composition methods. In this chapter, we have made concrete what each of these objects (H, E_1, g) are: sets of AG declarations D (which we will call modules). With this knowledge, we can discover more properties than are generally true for any system of language composition.

For instance, we now know that $H \lhd E_1 = H \uplus_\emptyset E_1$. Our composition method for attribute grammars is simply union of the declarations, and so both of these just indicate both the H module and E_1 module should be included in the composed result. The only difference is the *implication* that when we write $H \lhd E_1$ that E_1 is a module that depends upon H (and not vice versa.) Whereas with the latter we are suggesting (but not requiring, more of a connotation of our choice of notation) that the H and E_1 modules do not depend upon each other.

We can preserve that connotation, while transforming one notation to the other, with a little bit of work. If we take the E from $H \lhd E$ and separate it into two pieces:

those attribute grammar declarations that mention the host language in any way (call that E_{glue}), and those parts of the grammar that do not (call that E_{ind} for "independent"). (To actually perform this operation, we may have to use aspects to break production declarations apart to separate equations from each other.) The result is that $H \triangleleft E$ is equivalent to $H \uplus_{E_{glue}} E_{ind}$. This allows us to see these two operations (extension and composition with glue code) and being somewhat interchangeable.

The above implies we know something else about the operator $\uplus_g$. If we think about what $L_1 \uplus_g L_2$ would mean in terms of AG modules, we have two (perhaps independent) modules L_1 and L_2, with a third module g that necessarily depends on both of the others. That is, our "glue code" is just a module that imports two other modules (which are implied to be independent) and adds some declarations (that presumably make them work together somehow).

Finally, we look to our goal with language extension composition. We take the artifacts $H \triangleleft E_1$ and $H \triangleleft E_2$ and we wish to produce $H \triangleleft (E_1 \uplus_\emptyset E_2)$. The actual operation is quite simple: just union these three modules together, without adding any new declarations as "glue code". But so far, the problem is that we don't know enough about the result of doing so.

By virtue of (these days) rather ordinary language features like separate compilation and modular type checking, we can be confident that our automatically composed attribute grammar $H \triangleleft (E_1 \uplus_\emptyset E_2)$ is type correct. But we have no idea about attribute grammar well-definedness. Nor do we really have any more involved assurances about the behavior of the resulting program. These two problems are the primary subject of this thesis, and the subjects of the next two chapters.

4.6 Related work

In JastAdd [40] and Kiama[9], trees are represented as objects and attribute evaluation mutates the tree effectfully (either directly as in JastAdd or indirectly via memoization as in Kiama.) As a result, both of these languages lack a type distinction between the two kinds of trees. Instead, the user must remember to invoke a special copy method, analogous to our `new` expression, wherever a new undecorated tree is needed. These copy methods do not change the type of the tree, as our `new` operation does, resulting in a lack of the type safety that we have here. We avoid having to write `new` often by leveraging the type distinction to disambiguate, making our safety gains essentially "free."

UUAG[41] does not appear to support reference attributes, and so the type distinction is not exposed. In functional embeddings these type distinctions do occur naturally but typically having a different (internal, generated) name for the decorated view of the tree. AspectAG [10] is a sophisticated embedding into Haskell that naturally maintains the type safety we seek but at some loss of notational convenience. It also requires a fair amount of "type-level" programming that is less direct than the Silver specifications, and the error messages generated can be opaque.

Kiama and UUAG, by virtue of their embedding in functional languages, do support parameterized nonterminals and attributes. UUAG side-steps the attribute access problem of section 4.3.2 by simply not having reference attributes. As a result, attribute accesses expressions have only names on their left-hand side, not arbitrary expressions. As a result, all necessary information can be obtained by looking up the name, obtaining the explicit type from the declaration.

UUAG does not appear to support GADT-like productions, but we suspect it

could be easily extended to. Both UUAG and Kiama also support pattern matching on nonterminals. In UUAG, this is only supported for undecorated trees, and its behavior is identical to ordinary pattern matching in Haskell. In Kiama, pattern matching can extract decorated children from a production. But in both cases, use of pattern matching may compromise the extensibility of the specification, as new match rules cannot be added modularly.

Chapter 5

Modular well-definedness analysis

This chapter will present an analysis that ensures the well-definedness of an attribute grammar in a modular way. Traditional well-definedness (previously introduced in section 3.1.6) ensures that the attribute grammar is complete (no equations are missing) and non-circular (results can be computed for each attribute). However, the traditional approach is whole-program, and thus non-modular, meaning the well-definedness of a module may change in response to the introduction of more modules to the whole program. Thus, the well-definedness of even the host language may be at risk from the introduction of an extension (never mind how extensions may affect each other).

Resolving this problem is one of the two critical contributions of this thesis needed to ensure that the result of automatically composing language extensions will be sensible. This analysis ensures that there will be no missing or duplicate equations for synthesized or inherited attributes in a compiler composed of any set of extensions that satisfy the analysis. To accomplish this, we place modest restrictions on what extensions are allowed to do, and use these properties to show classical attribute grammar well-definedness.

This chapter is based on our previously published paper "Modular well-definedness analysis for attribute grammars"[77]. We have expanded this chapter significantly to be more precise about how the analysis works on the language Ag of the previous

chapter.

We begin by introducing modules to AG in section 5.1. Following that, we will discuss the concept of a modular analysis in section 5.2, specifically in contrast to the whole-program notion of the well-definedness analysis of attribute grammars. Finally, we will introduce (section 5.3) the notions of effective completeness and flow types, both essential to describing our goals and the analysis that will follow in this chapter. These sections constitute the background for the remainder of the chapter.

In section 5.4, we give an overview of how the analysis will work, describing the restrictions we will impose (in high-level terms) and how the accomplish our goals. Following that, in section 5.5, we describe in detail an algorithm for inferring flow types. (If the reader wishes to view flow types as declared rather than inferred, they can read subsections 5.5.1 and 5.5.3 describing flow information before skipping to the next section.) Finally, in section 5.6, we describe how to use the computed flow types to check for violations of the restrictions we introduced in the overview.

Concluding the chapter, we perform a small self-evaluation of the restrictions by applying the analysis to the Silver compiler itself in section 5.7. Following that, we discuss the approach and some of the concerns with extending this analysis to include non-circularity in section 5.8. And we conclude with some discussion of closely related work in section 5.9.

5.1 Modules

In chapter 4, we introduced the language AG (figure 4.1), a subset of the Silver programming language based on attribute grammars. This language covered attribute grammar declarations (D), equations, expressions, and types. We will now augment

module (or "grammar")
G ::= `grammar` n_g`;` $\overline{M}$ D
module statements
M ::= `imports` n_g`;` | `exports` n_g`;`

Figure 5.1: The module language for AG

it with a simple module language in figure 5.1.

A module (G) (in Silver referred to as a "grammar") consists of a declaration of its name (n_g), followed by some module statements, followed by the AG declarations that make up its body. The module statements indicate the dependencies of this module. The *direct imports* of a module are exactly those listed within the module using `imports`. Likewise, the *direct exports* of a module are those listed using `exports`.

An import indicates that this module requires another, and that other module's declarations will be visible to this modules implementation (D). An export *further* indicates that any module which imports this one will also need to see the declarations from another module. As a result, if module A imports module B and module B exports C, then A will also see C's declarations.

The set of modules exported by a module is the transitive closure over the export relationships of that initial module. That is, if we import module A, then we start with the set $\{A\}$ and iteratively expand this set based on the direct exports of any member of that set, until there are no more additions. This final set of modules is the full set of modules that are actually imported as a result. Thus, the set of dependencies that make up a module's normal environment follows that module's direct imports (i.e. non-transitively) and then transitively closes of all those module's exports. (Where the *transitive dependencies* of a module are those we obtain by closing over both imports and exports transitively.)

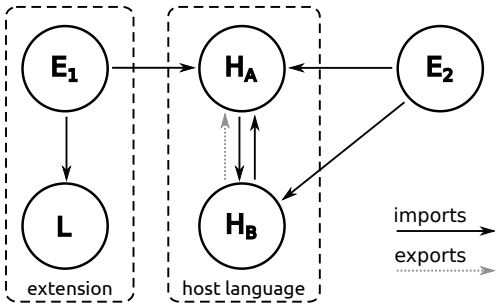


Figure 5.2: An example of a complicated module breakdown for a hypothetical host language and two extensions.

A host language might consist of multiple modules, and an extension may as well. In figure 5.2, we depict the host language as consisting of two modules (H_A and H_B). One of the extensions (E_2) consists of just one module. The other (E_1) extends the host language in addition to making use of another separate language (L). The extensions must depend on the host language somehow, but the host must not depend upon the extensions.

In this chapter, we will make a more fine-grained use of the module dependency relationships than just "host" and "extension." We will actually make restrictions at the module level. We will consider every module to be an "extension," to all the modules it imports, unless the imported module exports it back. In the figure, H_A is an extension to nothing (H_B exports it back), but H_B will be treated as an extension to H_A for our purposes. Likewise, the E_1 module extends *both* H_A and L.

We have chosen to include an exports relationship in order to make more precise some of the restrictions that will be introduced in this chapter. Strictly speaking we could do without introducing an "exports" relationship between modules, however some of our restrictions will require certain exports relationships to exist between two modules with related declarations. These restrictions could be instead be written

to require that these declarations are in the same module (a module always exports itself and so this would suffice), however this is overly restrictive and muddles the essential properties. A required exports relationship is directional (one module may export another but not vice-versa), but merging modules is not (both parts would then export each other). Preserving the directionality will help clarify exactly what is required.

Finally, a notational aid: we will use $\mathit{exports}(x, y)$ as a relation to determine the relationship between the modules x and y. That is, whether y is exported by x (or, reading the relation more directly, whether x exports y.) To make this relation easier to use, we will often directly apply it not just to modules, but to names of declarations from those modules. That is, we will write $\mathit{exports}(nt, \mathit{prod})$ to means "does the module that declares the nonterminal nt export the module declaring the production *prod*?" Similarly, we find it helpful to sometimes apply this relation to sets, for example $\mathit{exports}(\{x, y\}, z)$ means "is z exported by x **or** y?" (The reason we prefer disjunction will become clear once we begin describing the analysis later.)

5.2 Modular analysis

Our story for composing language extensions involves a host language and extensions that are each implemented as a set of modules. The extension modules may depend on the host, but no host module can depend on an extension module. With this setup, composition of several extensions with a host language simply means including more modules into the final compiler. This means the scope of what a language extension can change about a program is just what the inclusion of another module can affect about the whole program.

Each module has its own local perspective on the program, based on the modules it imports (i.e. directly depends upon.) We refer to this local knowledge as *sound* if it must be true, regardless of what other modules will be in the whole program. For instance, given a production declaration, we know its type signature, as this cannot later be changed by another module. However, given a nonterminal declaration, we do not necessarily know all of its productions, as other modules might introduce more. Our goal is to create enough sound local knowledge that we can perform a *modular analysis*: local checks that assure us of properties global to the program.

This kind of modular analysis is hardly unheard of: type checking is one example. Indeed, for any language with separate compilation, all analyses involved must be modular. A modular analysis is in contrast to a *whole-program* analysis, like the attribute grammar well-definedness analysis we have previously described (section 3.1.6). A whole-program analysis may change as a result of including new modules.

Copper's *modular determinism analysis* for context free grammar fragments is a modular analysis in this sense. It is run independently by each language extension developer to verify that their extensions to the grammar pass. When a set of verified extensions are composed, it will result in a deterministic grammar from which a conflict-free LR(1) parse table can be constructed [67]. Formally, this was expressed as

$$(\forall i \in [1,n].\mathit{isComposable}(CFG^H, CFG_i^E) \wedge \ \mathit{conflictFree}(CFG^H \cup \{CFG_i^E)\})$$
$$\implies \mathit{conflictFree}(CFG^H \cup \left\{CFG_1^E, \ldots, CFG_n^E\right\})$$

Where *isComposable* is the modular analysis, and *conflictFree* is the traditional (whole-program) LR parser construction analysis. Note that each extension grammar

CFG_i^E is tested in isolation (i.e. only with respect to the host language grammar CFG^H,) but the conclusion is about all extensions composed together. Of course, this analysis puts some restrictions on the type of syntax that can be added to a language as a composable language extension, but we have found these restrictions to be natural and not burdensome [67].

The primary contribution of this chapter is a modular analysis for the well-definedness of attribute grammars. This analysis has the same basic shape as the Copper modular analysis:

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

Where *modComplete* is our new modular analysis, and *complete* is the traditional attribute grammar completeness analysis (or rather, the *effective completeness* variation discussed in the next section.) As a result, we know that the host language composed with any number of extensions will have a complete attribute grammar. Or in other words, it will not have any missing equations.

Well-definedness typically also includes, in addition to completeness, an analysis to ensure non-circularity. We will begin by considering only completeness, and discuss extending this to circularity (as well as some reasons we perhaps should not do so) in section 5.8. Additionally, the traditional AG notion of well-definedness does not concern itself with duplicate declarations, but this is merely because the traditional definition of AGs is not as a programming language. Instead, it assumes such things as "a function from nonterminals to a set of attributes that occur on it," and so duplicates are impossible by definition of "set". And so our modular well-definedness analysis, in addition to concerning itself with completeness (the presence of needed equations),

will also concern itself with duplicates (the non-presence of repeated occurrences or conflicting equations).

As a result, the non-expert can direct Silver to compose host and extension specifications (that pass this analysis) knowing that the resulting attribute grammar will be complete. Thus, the entire class of attribute grammar errors that could cause composition to fail can be solved by the extension developers, and this burden will not fall on the non-expert who ultimately does the composition of the extensions. This ensures composition will succeed, and sets the stage for the next chapter, where we go further and ensure the composed compiler will behave sensibly.

5.3 Effective completeness and flow types

The traditional completeness property for attribute grammars simply requires, for every synthesized attribute that occurs on a production's nonterminal, there must be an equation giving it a value. And for every inherited attribute on each child, there must be an equation giving it a value.

A problem identified when forwarding was introduced [21], but also existing for higher-order attribute grammars, is the inconvenience of requiring *all* inherited equations to exist. Frequently, only a subset of synthesized attributes are demanded from a subtree, which in turn only require a subset of inherited attributes to be provided. With higher-order attributes, a tree may be decorated for the sole purpose of accessing only one synthesized attribute, such as a transformation. If the transformation does not require some inherited attributes, it is pointless to supply them. With the introduction of forwarding, a production may only use its own children to compute a pretty-printing, for example, but rely on the forward tree for everything else (such as

its type checking or translation.) This production would be required to supply all of its children with inherited attribute equations that are never used, merely to pretty print the tree.

An amended notion of *effective completeness* can be used instead. While every synthesized attribute still needs an equation, we require only that all inherited attributes *needed to compute any* ***accessed*** *synthesized attribute* are supplied, instead of simply all of them outright. With an effectively complete attribute grammar, we can still compute non-circularity in the same manner as with a complete one: if these missing equations are never demanded, they will never have an effect on how information flows around in the attribute grammar.

As a bonus, we have a hope of constructing a modular analysis for effective completeness, whereas this is not possible for traditional completeness. Any extension can introduce new inherited attributes, and any extension can contain new *decoration sites*, which are those places where a tree is supplied with a set of equations that compute its inherited attributes, resulting in a decorated tree. (For example, every child of nonterminal type is a decoration site within that production.) With traditional completeness, we have no real hope of ensuring two independent extensions resulting in a complete attribute grammar. With effective completeness, there is a possibility: perhaps all new decoration sites will not need equations for the new inherited attributes from different extensions.

In figure 5.3, we show an example of an effective completeness problem, with two extensions to the boolean grammar of figure 3.5. This example shows us the kind of problem we need to avoid, as these two independent extensions are fine in isolation, but their composition results in a necessary but missing equation. We see that the `iff` production has introduced new decoration sites: it accesses `eval` on its

var extension

```
inherited attribute env::Env;
attribute env occurs on Expr;

aspect and
e::Expr ::= l::Expr r::Expr
{ l.env = e.env;
  r.env = e.env;
}
aspect or
e::Expr ::= l::Expr r::Expr
{ l.env = e.env;
  r.env = e.env;
}
aspect not
e::Expr ::= s::Expr
{ s.env = e.env;
}
```

Details of `Name` and `Env` types omitted for brevity, as well as the `lookup` function.

```
production var
e::Expr ::= n::Name
{
  forwards to
    lookup(e.env, n.name);
}
```

the iff extension

```
production iff
e::Expr ::= l::Expr r::Expr
{ e.eval = l.eval == r.eval;
  forwards to and(implies(l, r),
                  implies(r, l));
}
```

Figure 5.3: An example of two extensions to the boolean grammar of figure 3.5. The `var` extension introduces a new forwarding production and inherited attribute. The `iff` extension is reproduced from the original figure for reference, and is now considered an extension. (The var extension will turn out to violate our modular analysis.)

children, and so we must create decorated trees for those children. We also see that the `var` extension introduces a new inherited attribute. However, there is no equation supplying `env` to the children of an `iff` node, and so if a `var` node appears below a `iff` node, we will run afoul of this missing equation. To see precisely how this could happen, we need some additional tools.

5.3.1 Flow types

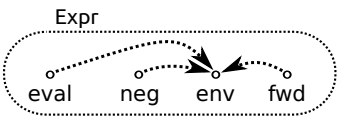
The traditional attribute grammar circularity analysis computes graphs representing the potential ways information can flow through trees. Each synthesized attribute on

a node in the tree could, in some way, ultimately depend on some of its own inherited attributes, but these dependencies are usually non-local: they arise because values gets passed down into and computed from child subtrees. To accomplish this analysis in a modular way, and on AG, we need to make several changes to how things are traditionally done.

Unfortunately, in order to describe how we will construct flow graphs, we need to explain the aims of the modular analysis. And in order to describe the aims of the analysis, we need to discuss the kinds of dependencies that can arise in our altered flow graphs. To resolve this organizational conundrum, we will begin by introducing the concept of a *flow type* independently of production flow graphs. In section 5.4, we will describe the aims of the analysis in terms of flow types. And then in section 5.5.1, we will describe in detail what our production flow graphs look like. (Some readers may find it helpful to look ahead at the picture of production flow graphs in figure 5.7 to get a sense of what they look like, before reading on in this section.) However, this means for the moment we have to describe the concept of flow types rather abstractly.

In the traditional circularity analysis, each nonterminal is associated with a *set* of flow graphs showing the different ways in which the synthesized attributes on that nonterminal may depend upon the inherited attributes on that nonterminal. To determine effective completeness, we need some of this information, but not all of it. It's not necessary to know the different ways things depend on each other, only whether they might. As a result, we don't need a set of different flow graphs.

One of the nice properties about these nonterminal flow graphs is their lattice structure: we can look at each flow graph as a set of edges and use the subset relation as a partial ordering on graphs. One property that is traditionally used as an optimization[78] is that we can simply drop any graph that is a subgraph of another



$$ft_{Expr}(eval) = \{env\}$$
$$ft_{Expr}(neg) = \{env\}$$
$$ft_{Expr}(fwd) = \{env\}$$

Figure 5.4: A flow type (as both a function and as a graph, where edges denote dependencies rather than direction of information flow) for the boolean language including the `var` extension from figure 5.3.

graph already in the set. That is, if we already have a graph $\{a \to i, b \to i\}$ in the set then it will never be necessary to introduce $\{a \to i\}$, as this is "covered" by the first. We aren't going to learn any new flows of information by omitting information flow we already know about, and so it's not necessary to retain them.

Using this lattice structure, we can simply take the join over all the flow graphs for a nonterminal, producing a *flow type* [79]. (The join of the set of graphs, in this case, is simply the union of their sets of edges.) A flow type can also be thought of as a function $ft_{nt} : syn \to \{inh\}$ that defines, for a nonterminal, what inherited attributes each synthesized attribute that occurs on that nonterminal may depend upon. This representation takes advantage of the fact that these graphs are bipartite: the only kind of edge allowed is the dependency of a synthesized attribute on an inherited one (never inherited to synthesized, or synthesized to synthesized, etc.) We will freely conflate both of these ways of looking at a flow type: graph and function.

If we do not care about non-circularity, we can drastically improve the efficiency of the flow algorithm by computing flow types directly, instead of via the join of the set of nonterminal flow graphs. (We leave concerns about circularity to section 5.8.) The size of the set of flow graphs can grow exponentially with the number of attributes, and but there is always only one flow type for a nonterminal. All we do to accomplish

this modification of the algorithm is exploit the lattice structure along the way, by simply always updating the single flow type instead of adding new graphs to the set. (Details of this algorithm are deferred until section 5.5.5, but its general shape is a typical "iterate until fixed point" algorithm updating the current flow types.)

For the purposes of our modular analysis, we must make an additional extension to the flow type for each nonterminal. In addition to tracking dependencies for each synthesized attribute, we wish to track them for the forward equation as well. (In the equations of AG (Q), the forwards equation is `forwards to` E .) We will soon consider restrictions on flow types, and some of these restrictions will involve the set of inherited attributes that may be used in computing the tree a production forwards to. We will write $ft_{nt}(fwd)$ to refer to the dependencies potentially necessary to evaluate all forward equations for the nonterminal nt.

In figure 5.4 we show an example flow type for the boolean language with the `var` extension. How we actually compute this flow type is the subject of a later section (5.5.)

5.4 Overview of the analysis

In this section, we will sketch a high-level understanding of what the analysis does, and how it will work. We will begin by describing just what it is that extensions will not be allowed to do as a result of this analysis, and how these restrictions accomplish our goals. Finally, we will motivate a structure for how this analysis will proceed that is slightly different from what we originally suggested a modular analysis looks like.

5.4.1 Restrictions imposed by the analysis

The analysis has two separate but related phases:

1. We ensure certain exports relationships exist for each declaration. This is sufficient to ensure that all synthesized equations are present (and that no duplicate equations exist) using only local checks.

2. We examine flow types for the host (ft^H_{nt}) and extension (ft^E_{nt}), and make sure certain relationships exist between them. This is sufficient to ensure that we only need local checks for all necessary inherited attribute equations.

The first phase is similar to a requirement in Haskell for instances of typeclasses, called the *orphaned instances* rule, and so we call these orphaned declarations. The purpose of this sort of rule is to ensure no duplicates could possibly exist. Suppose we have three separate modules: N which declares a nonterminal, A which declares an attribute, and O which declares an occurrence of this attribute on the nonterminal. To write this declaration, O must import both N and A. We wish to ensure there could never be duplicate occurrences, so we require that $\mathit{exports}(\{N, A\}, O)$. (Recall that while *exports* is a relation on modules, we take the notational liberty of applying it to declarations, meaning the modules containing those declarations.) In other words, one of N or A (or both) must export O. In order to write another occurrence besides the one in O, we must import both modules, and so as long as one of them exports the existing O, a local check is sufficient to globally forbid duplicate declarations.

We repeat this rule for equations. Suppose we have the modules O, P which declares a production, and E which declares an equation for this attribute on this production. We likewise require $\mathit{exports}(\{P, O\}, E)$. Thus we can forbid duplicate

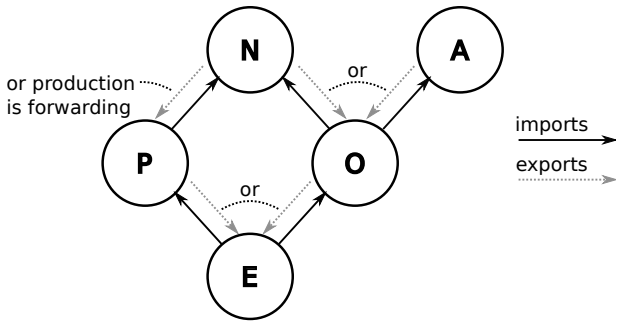


Figure 5.5: A summary of the import/export relationships required between the modules that contain related attribute grammar declarations (nonterminal, attributes, productions, occurs, and equations.)

equations.

Finally, a slight variation of this rule for productions. A production constructs a particular nonterminal, and so must import N (which we assume is the LHS nonterminal for the production in P.) Either it must be the case that $\mathit{exports}(N, P)$, or the production must be forwarding. As a result, we have a fixed set of non-forwarding productions for each nonterminal, determined by N. Any extension production must (as a result) be a forwarding one.

These restrictions are summarized visually in the graph in figure 5.5. Remember that actual structure can always be simpler than we depict here, because these modules can always be the same module. Since a module always exports itself, and our restrictions are always about lack of an exports relationship, a single module importing no other module can never run afoul of our restrictions. (In fact, this will remain true even as we introduce additional restrictions later.) As a result, a classically well-defined host language consisting of only one module is always modularly well-defined.

There is another important property these restrictions accomplish. The O module must import N, which must export all P containing non-forwarding productions. Because all equations E must be exported by P or O, we are able to see (from O) all *synthesized* attribute equations that must be present for that attribute on that nonterminal. As a result, checking for the presence of equations locally from O is sufficient to ensure completeness of synthesized equations. Further, this also allows us to implement pattern matching exhaustiveness checking normally, as the modified pattern matching semantics means we need only cover non-forwarding productions.

The second phase of our analysis involves computing flow types. One way of thinking about this is to perform flow type computations for each module. For each nonterminal nt being imported, we compute ft^H_{nt} on the set of imported modules, and ft^E_{nt} again now with the extension module now included. (We impose no flow restrictions on those nonterminals declared in the extension module, only those that are imported.) We have several requirements on the values we compute for these.

First, for every synthesized attribute s that occurs on nt in H, it must be the case that $ft^H_{nt}(s) = ft^E_{nt}(s)$. It's not possible for extensions to remove things from the flow type, and it must also be the case that extensions do not add to the flow type of an attribute occurrence from the host language. As a result of this restriction, we always know what inherited attributes every synthesized attribute access may need.

When we return to look at the `var` extension of figure 5.3, and the flow type computed for it in figure 5.4, we can see a violation of this restriction. Originally, we would have computed $ft_{Expr}(fwd) = \emptyset$, but the `var` extension has introduced a new dependency on `env` (and for the other two synthesized attributes as well.) Other extensions cannot necessary know of this new depedency, and as a result may not have an equation for it. Thus, this extension is not modularly well-defined (despite

appearing well-defined in isolation,) and this latent problem is exposed by the `iff` extension again shown in figure 5.3.

This restriction is almost enough to ensure that all necessary inherited equations are present, but there's a little more we need. In addition to host language synthesized attributes, we repeat this requirement for the forward dependencies we are tracking. That is, for every nt in H, $ft^H_{nt}(fwd) = ft^E_{nt}(fwd)$. This is necessary to ensure that extensions do not use new inherited attributes in their forward equation, and as a result cause those to pollute the dependencies of synthesized attributes through forwarding's implicit equations.

Note that we are only talking about nonterminal and occurrence declarations from the imported module H. New nonterminals and synthesized occurrences introduced by the extension are not subject to the above restrictions (except when writing extensions to extensions, of course.) However, there is one restriction that does apply to new synthesized occurrences.

It must be the case that for each new occurrence of a synthesized attribute s in the extension on a nonterminal nt **from** H, that $ft^H_{nt}(fwd) \subseteq ft^E_{nt}(s)$. This one is slightly subtle. Consider two independent extensions, both introducing forwarding productions with equations that make full use of their allowed dependencies. Suppose one introduced a synthesized attribute that violated the above rule. For example, $ft_{nt}(s) = \emptyset$ while $ft_{nt}(fwd) = \{env\}$. We would now suffer a missing inherited equation if we attempt to access this attribute (which claims to require no inherited attributes) from a tree rooted in the forwarding production from the independent extension. We need that inherited attribute to compute the forward tree, which would then have the equations that give a value to this attribute. Thus, this minimum requirement for the dependencies of extension attributes on host nonterminals.

```
N_A   nonterminal A;
N_B   nonterminal B;
A_S   syn S :: T;
A_I   inh I :: T;
O_S   attr S occurs on B;
O_I   attr I occurs on B;
P_A   production pa
      l::A ::= r::B
      {
E_I    r.I = ...;
cloud  ... r.s ...
      }
P_B   production pb
      l::B ::=
      {
E_S    l.s = l.i;
      }
```

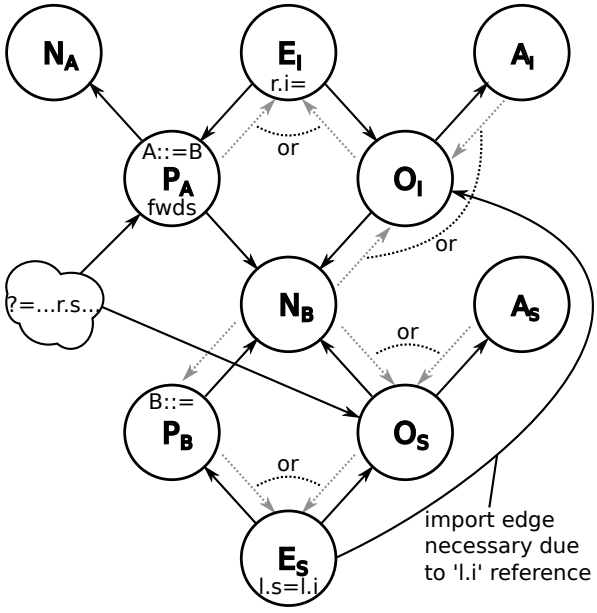


Figure 5.6: A worst-case module imports/exports diagram, showing the relationship between an access of a synthesized attribute and the required location of an inherited equation it may need to exist. On the left: AG code that, when exploded into multiple modules, would give rise to this graph.

5.4.2 Reasoning about the presence of inherited equations

While seeing how synthesized completeness can be accomplished was fairly straightforward, understanding how we are able to check for effective inherited completeness as a result of these restrictions is slightly more involved. We illustrate this with a worst-case scenario.

In figure 5.6, we show (as a cloud-bubble) an access to a synthesized attribute. This figure depicts two nonterminals A and B, declared in corresponding modules N_A and N_B, and with corresponding productions within P_A and P_B. The equation of interest is for the production for A, which we claim is forwarding in order to minimize the exports relationships. We make no assumptions about what this equation is for

(perhaps some other, not-shown attribute,) which is why we depict it as a cloud bubble. This equation accesses s on a child which is of nonterminal type B. The flow type corresponding to this access ($ft_B(s)$) indicates that it depends upon an inherited attribute (i) as a result of the equation in E_S. We must ensure that we are able to discover this equation within the module E_I starting from the cloud bubble.

One important constraint not shown in the diagram is that one of N_B or O_S must export E_S. If one of these does not exist, then the equation in E_S would not be permitted to affect the flow type of that synthesized attribute.

Our goal is to arrive at the inherited equation while following only edges that must exist. We begin with the required imports from the access (the cloud bubble) of $\{P_A, O_S\}$, as to write an equation depending upon the child's synthesized attribute, we must of course be able to aspect this production and access that attribute. By following the import from P_A, we can expand this set to also include N_B. So far we are sure of at least $\{P_A, O_S, N_B\}$.

Because we have both N_B and O_S, we can conclude we must have E_S. This was the non-visually represented constraint just mentioned: otherwise the equation could not have affected the flow type. We can follow the import from E_S to O_I, giving us $\{P_A, O_S, N_B, E_S, O_I\}$. Then because we have both P_A and O_I we must have E_I, completing the proof. The inherited equation demanded by an attribute access must always exist somewhere within the transitive dependencies of a module.

It is worth noting, before we go further, that this can be simplified. Merely noting that the equation exists somewhere in all transitive dependencies is sufficient, but it means we need more information that just what the normal environment would provide. If we wanted to use the normal environment to perform these checks, it would suffice to require that an exports relationship exist from a synthesized attribute

occurrence to all inherited attribute occurrences that are in its flow type. That is, O_S would have to export O_I. This means that starting with the imports ($\{P_A, O_S\}$) and following *only exports* arrows successfully gives us $\{P_A, O_S, O_I, E_I\}$. We opted for the transitive dependencies approach merely because we thought it would be odd to require extensions to export their host language, and looking through transitive dependencies presents no problems for composing language extensions. This approach also keeps the analysis sensible even when export relationships do not exist (i.e. that "exports" means they are the same module, and thus extensions could never export the host language.)

5.4.3 Integrating flow and effective completeness analysis

Although we have presented modular analysis as an extra set of local restrictions (*modComplete*) that ensures a whole-program analysis (*complete*) will be preserved, we will not present our algorithms as two separate components. The trouble with this presentation is that it implicitly assumes non-circularity of module dependencies. We have been forced by practical concerns to accept circular dependencies between modules, and so our algorithm is designed to handle this.

We can do this simply enough by integrating the modular restrictions and the traditional completeness analysis. In the presentation above, we describe a restriction on flow types as having computed two flow types: ft^H_{nt} and ft^E_{nt}. What we will do instead is perform a single flow type computation over all modules being compiled, but with modifications that allow us to mark some equations as not being allowed to affect flow types.

These two approaches will have the same behavior whenever modules do not have cyclic dependencies. However, if two modules import each other (but do not

export each other), the new approach will correctly constrain each module in how flow types can be affected. Whereas if we had directly applied our previously suggested approach, the flow types computed would always be equal (as they both would include the same set of modules.)

Changing the flow analysis to accommodate equations that aren't allowed to affect flow types, however, merges these two separate things (our modular restrictions and the usual completeness check) together into one analysis. We will see very shortly exactly how this modification is done.

In principle, however, we wish to point out that there is an alternative to this approach. Although we attempt to infer flow types for an attribute grammar, we could instead change the language so that occurs declarations provide an explicit flow type for each synthesized attribute. We could then simply take these as authoritative and skip parts of the inference step described the next section. However, inference is quite helpful and allows us to apply this analysis to traditional attribute grammars.

5.5 Flow type inference

We will present the analysis in several stages. First, we will introduce production flow graphs, a necessary internal piece of how the flow analysis is done. We have previously talked about flow graphs associated with a nonterminal, which just contains vertexes for the synthesized and inherited attributes on that nonterminal (plus an extra node for tracking forward equation dependencies.) A production flow graph will expand this to include many more vertexes, including for all attributes on all children of that production (and more that we will describe in detail shortly.)

We will then see how to compute, given an Ag expression, a set of vertexes

within a production flow graph that the expression depends upon. Next, we will compute an intermediate representation (that we call *FlowDef*) that abstracts away from Ag to just that information relevant to a flow analysis. Finally, we will use this information to construct "static production flow graphs" which are the inputs to a slightly modified version of the traditional attribute grammar flow analysis. The result of all this is the inference of flow types for a set of modules, and these flow types will respect our restrictions on what modules can affect which flow types.

5.5.1 The structure of production flow graphs

In order to compute how information flows around in an attribute grammar, we begin with *static production flow graphs*. These are "static" in the sense that they do not yet contain any inter-production flow information, as they are the starting point for the flow analysis. They only show direct dependencies that are the result of equations within a production. We compute one of these graphs for each production.

In figure 5.7, we show static production flow graphs for some of the productions in figure 5.3. We are about to dive into all the details necessary to fully understand these graphs, but it should be somewhat obvious what each arrow represents. For example, the production `and` has the equation `e.eval = l.eval && r.eval`, and so we draw two arrows from `e`'s `eval` vertex, to the `eval` vertex of both `l` and `r`. The graph shows some arrows that are the result of implicit equations, too, not just explicit ones. For instance, because `var` does not have any synthesized equations, we see arrows from its synthesized attributes to the corresponding vertexes within `forward`.

At this point, we need some terminology to aid us. In figure 5.8, we show the labeled components of a flow graph. We refer to each vertex in the graph as a *flow*

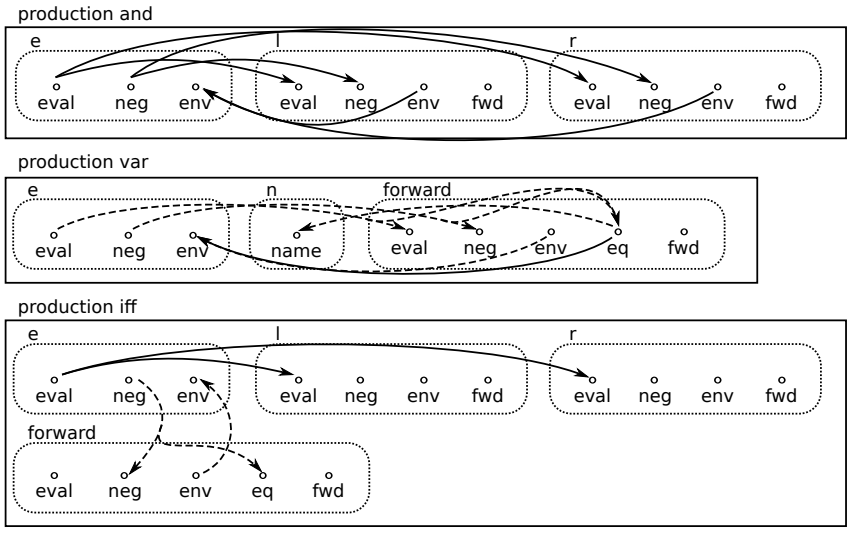


Figure 5.7: Some flow graphs for the grammar of figure 5.3. Arrows show direct dependencies.

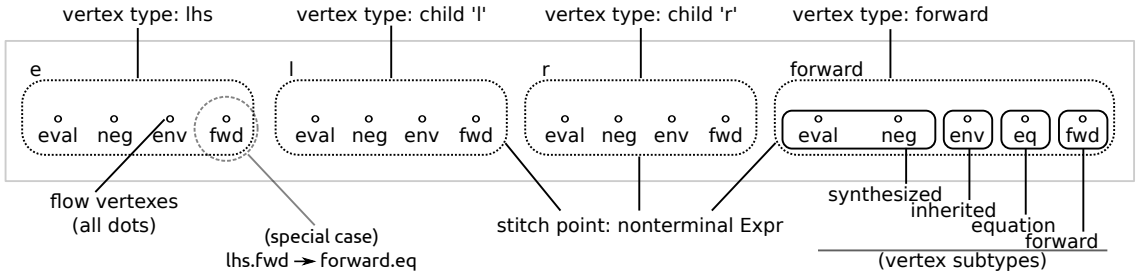


Figure 5.8: The `or` production, minus edges, with the different components of a static production graph labeled.

vertex. Each flow vertex belongs to a *vertex type*, for example the left-hand side vertex type labeled 'e' for each production in the figure.

We can classify vertex types into a few groups: left-hand side, child, forward, local, and anonymous. These correspond to the different ways an undecorated tree can be decorated within a production (plus the left-hand side, which is a special case.) Note, for example, that a vertex type is specific to which child; i.e. "child r" is a vertex type, while just "child" is it general classification. Every flow vertex

belongs to a vertex type, and we can also classify these different flow vertexes within a type: synthesized, inherited, forward, and equation. (The latter two we will explain shortly.) So in the production `and`, `l.eval` corresponds to the synthesized flow vertex `eval` within the child vertex type `l`.

The static production graph also comes with a set of *stitch points*. These are simply the locations where we might later introduce new edges as a result of discovering what sorts of indirect dependencies may exist in the attribute grammar. Every vertex type (except for the LHS) is a stitch point, and we will often refer to these as *decoration sites* because these are where undecorated trees are supplied with inherited attributes. (The LHS is neither a decoration site nor a stitch point, because a production does not supply itself with its own inherited attributes.) Recall that we intend to compute flow types, which indicate the dependencies of synthesized attributes (and the forward equation) on the inherited attributes. And so later on, when we add flow type information to a static production graph, we will create internal edges within each stitch point. These will go from the synthesized flow vertexes (and the `fwd` flow vertex) to the inherited flow vertexes within the same vertex type.

Not all vertex types are treated in the same way, as we've already seen since LHS is uniquely not a stitch point. We summarize some of these different properties in figure 5.9. Another major difference is that the LHS and children do not have an "equation" flow vertex (labeled `eq`.) An equation flow vertex corresponds to the equation that constructs the related object. For example, a "`forwards to` E" equation constructs a tree that the production forwards to. Thus, the `forward` vertex type has a flow vertex labeled `eq` that shows the dependencies of the expression E. This is repeated similarly for locals (`local x::T =` E) and anonymous decoration sites (`decorate` E `with {...}`). But the LHS and children do not have any such

	Equation	Forward	Stitch point
LHS	No	(redirected)	No
Child	No	Yes	Yes
Local/Forward/Anonymous	Yes	Yes	Yes

Figure 5.9: Properties of flow vertex types. For each vertex type, whether it has equation and forward flow vertexes, and whether it is a nonterminal stitch point.

equation, and so do not have this flow vertex.

As a simple example, back in figure 5.3, the `var` production uses `env` from its LHS (`e`) in its forward equation. As a result, we see in figure 5.7, the production flow graph for `var` has a direct solid line from `forward.eq` to `e.env`.

Finally, there is one more special case for the LHS vertex type. All other vertex types have a `fwd` flow vertex that represents the dependencies necessary to compute its forward tree. The LHS, however, does not really need such a thing, because it exactly corresponds to the `eq` vertex of the `forward` vertex type. Whether this extra vertex on the LHS is dropped, or an edge is always present from the LHS `fwd` to the forward `eq` is an implementation detail. The important point is simply that this is the only instance of two flow vertexes being identified.

The `fwd` flow vertex of the `forward` vertex type is not even a special case. Dependencies on a `fwd` flow vertex can arise as a result of pattern matching, as recall we must be able to evaluate the forward equations of the productions we are matching against. As a result, if we pattern match against the forwarded to tree directly (for example), this would emit a dependency on `forward`'s `fwd` vertex.

So far this description of static production flow graphs is similar to that used for a whole-program analysis on Ag. However, recall that we need to modify the way we compute flow information in order to handle cyclic dependencies between modules.

To handle this, instead of allowing all equations to affect flow types and computing them twice, we will directly limit some equations so that they are unable to affect the flow type. This allows us to compute flow types once, and then return to each equation to discover violations later.

We accomplish this by have two different sets of edges in the production flow graph: ordinary edges and *suspect edges*. Suspect edges are edges that should not be able to have an effect on corresponding flow types. A suspect edge originates only from the LHS synthesized vertexes or from the equation vertex of the forward vertex type (i.e. those things that we track the potential dependencies of in the flow type.) We already showed some suspect edges by drawing them with dashed lines back in figure 5.7. Because the `var` and `iff` productions are in extensions and not exported by the nonterminal `Expr`, the marked edges are suspect.

We must remember suspect edges and cannot simply drop these edges from the graph entirely because they *should* have an effect on the flow types of extension synthesized attributes. Consider the following simple extension production:

```
production extension
e::HostNT ::=
{
  e.hostSyn = ... e.inh ...;
  e.extSyn = ... e.hostSyn ...;
  forwards to ...
}
```

We would have (from bottom to top) an ordinary edge from `extSyn` to `hostSyn` on the LHS, and we would generate a suspect edge from `hostSyn` to `inh`. If we simply discarded these suspect edges, we would end up inferring no inherited dependencies for `extSyn`, which we can see is wrong.

Thus, a *static production flow graph* is a tuple consisting of:

- The left-hand side *nonterminal*.
- A directed graph of ordinary edges between *flow vertexes*.
- A set of *stitch points*.
- A set of *suspect edges*.

So far we have described one kind of stitch point, a *nonterminal stitch point*, where a flow type for a nonterminal causes new edges to be introduced into the production graph. However, to better account for pattern matching, we must introduce a second kind of stitch point: a *production projection* stitch point. Where a nonterminal stitch point produces internal edges *within* a vertex type according to its flow type, a projection stitch point produces edges (for inherited attributes only) from one vertex type to another vertex type according to another production flow graph.

When pattern matching, we always have a vertex type for the scrutinee. (If a particular scrutinee does not have a vertex type, we will create an anonymous one for it. This will be shown in a precise way later on.) We also have a vertex type for each pattern variable (n.b. each pattern *variable*, not pattern. So `pair(x, y)` gets two vertex types: one for `x` and one for `y`.) We will also create a projection stitch point between each pattern variable vertex type[1] and the scrutinee vertex type. These edges indicate how the inherited attributes for pattern variables are ultimately related to the inherited attributes given to the scrutinee of a pattern matching expression. A pattern variable corresponds to a child of another production, and so we project each

[1]Although we're calling these "pattern variable" vertex types to be clear here, these are just anonymous vertex types that are also used for `decorate` expressions.

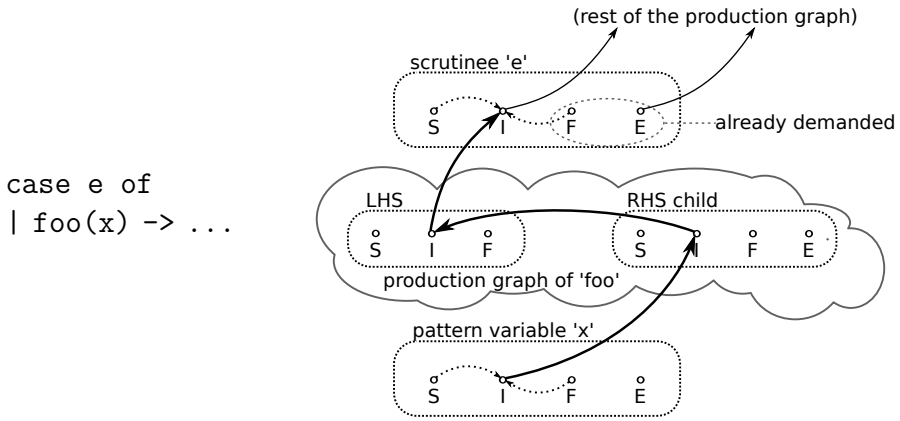


Figure 5.10: An illustration of how a projection stitch point creates edges from a pattern variable's inherited attributes to the scrutinee's inherited attributes. The dotted edges are from the nonterminal stitch point (according to the flow type) in each vertex type, and the solid path from `x.I` to `e.I` corresponds to the newly introduced edges.

inherited attribute for the pattern variable to the corresponding vertex in the other production, follow these back to the LHS inherited attributes in that production, and then emit these as edges to the inherited attributes of the scrutinee. This is illustrated visually in figure 5.10. In this way, we compute the same flow information that would have been computed had we translated the pattern matching expression into attributes.

And so, we have a taxonomy of vertex types and flow vertexes that make up a production flow graph, but just two general kinds of stitch points. To get a general sense of what a production flow graph will consist of, we summarize the origins of the different stitch points that will appear in a typical one:

- Every vertex type (except the LHS) is also a nonterminal stitch point. This means every child and local, and the forward.
- Each `decorate` expression yields an anonymous vertex type (and thus a nonterminal stitch point.)

- Some pattern matching expressions will create an anonymous vertex type for the scrutinee, if that expression does not already have one. (The details of this will be seen later.) These are likewise nonterminal stitch points.
- Every pattern variable will have an anonymous decoration site and associated nonterminal stitch point.
- Each pattern variable will *also* have a projection stitch point, relating it to the scrutinee vertex type.

Finally, there is one last subtle detail of vertex types that merits attention. For the child and local vertex types, we are going to make an assumption in order to simplify the presentation of the remainder of this chapter. We will always assume they have nonterminal type. If a child does not have nonterminal type (perhaps it is a decorated type, or a function type, or some other primitive type), then in a practical implementation *that child has no vertex type.* A vertex type does not make sense for a child of type (for example) `Integer`: it doesn't have attributes, so what are the vertexes? For locals, we still need to worry about the dependencies its defining equation might have, and so for Silver we tolerate a degenerate kind of local vertex type that has no stitch point and contains only an `eq` node. However, including these in the presentation here just introduces a tedious proliferation of very simple special cases, so we skip them to focus on the essential ideas.

5.5.2 Flow dependencies of expressions

Our next task in developing this flow type inference algorithm is to compute the flow vertex dependencies of expressions. So far we have looked at expressions like `l.eval`

`&& r.eval` and discovered the two obvious flow vertex dependencies. However, not all expressions are so trivial. Even this simple example is hiding a significant subtlety.

Since AG allows us to take a reference to a decorated node, pass that around, and then access an attribute from it. We must come up with some solution to the problem of determining what inherited attributes must be given to (and can be accessed from) a reference. To solve this problem, we introduce a function *ref* that takes a vertex type, and maps it to a fixed "blessed set" of inherited attributes for that nonterminal. In Silver, we define this set as all inherited attributes with an occurrence exported by the nonterminal declaration. Other definitions are possible but there are two essential features for how *ref* must behave. First, all modules must agree on the answer, so it should be determined by the module that declares the nonterminal. And second, the reference set should be a superset of $ft_{nt}(fwd)$. In other words, we should always be able to compute how trees forward, given a reference. (Without this property, references are vastly less useful, since we could never do things like access an extension synthesized attribute from one.)

Returning to our supposedly simple example, the `l` subexpression is a reference to a child, and so we should be emitting dependencies on all inherited attributes given by *ref*(`l`). This is, after all, an expression that references that child. But we're then going on to immediately access `eval`, which might depend on much less that what is necessary to take a reference. If we're to find accurate dependencies, then we cannot just map an expression to a list of flow vertexes, we need more information.

We define a function on AG expressions that produces two pieces of information. First, if the expression is a direct reference to a vertex type, it computes which. Second, it computes a list of other vertexes the expression requires in order to be evaluated. There are three general ways we will make use of this information:

$$
\begin{aligned}
\mathscr{D}(x) &= \begin{cases} \langle \mathit{vertexType}(x), \mathit{ref}(N) \rangle & \text{where } x :: N \\ \langle \mathit{none}, [] \rangle & \text{if } x \text{ has no vertex type} \end{cases} \\
\mathscr{D}(E(\overline{E_a})) &= \langle \mathit{none}, \mathscr{D}(E) +\!\!+ \overline{\mathscr{D}(E_a)} \rangle \\
\mathscr{D}(\backslash \overline{x :: T} \texttt{ -> } E) &= \langle \mathit{none}, \mathscr{D}(E) \rangle \\
\mathscr{D}(E \,.\, a) &= \begin{cases} \langle \mathit{none}, v.\mathit{for}(a) +\!\!+ v.\mathit{eq} \rangle & \text{if } \mathscr{D}(E) = \langle v, d \rangle \\ \langle \mathit{none}, d \rangle & \text{if } \mathscr{D}(E) = \langle \mathit{none}, d \rangle \end{cases} \\
\mathscr{D}(\texttt{decorate } E \texttt{ with \{ } \overline{A} \texttt{ \}}) &= \langle \mathit{anonVertex}(\nu), \nu.\mathit{eq} +\!\!+ \mathit{ref}(N) \rangle \text{ where } E :: N \\
\mathscr{D}(\texttt{new } E) &= \begin{cases} \langle \mathit{none}, v.\mathit{eq} \rangle & \text{if } \mathscr{D}(E) = \langle v, d \rangle \\ \langle \mathit{none}, d \rangle & \text{if } \mathscr{D}(E) = \langle \mathit{none}, d \rangle \end{cases} \\
\mathscr{D}(\texttt{case } E \texttt{ of } \overline{P \texttt{ -> } E_p}) &= \langle \mathit{none}, \overline{\mathscr{D}(E_p)} +\!\!+ \\
&\qquad \begin{cases} v.\mathit{fwd} +\!\!+ v.\mathit{eq} & \text{if } \mathscr{D}(E) = \langle v, d \rangle \\ d & \text{if } \mathscr{D}(E) = \langle \mathit{none}, d \rangle \end{cases} \rangle
\end{aligned}
$$

Figure 5.11: Computing vertex dependencies from expressions.

- A subexpression references a vertex type, and its parent in the tree is able to take advantage of that fact. (e.g. `child.syn`) In which case, the resulting dependencies will depend only on the vertex type.
- A subexpression references a vertex type, but its parent cannot use that information. (e.g. `foo(child)` where `foo` is a function taking a decorated type.) For this case, the vertex type is ignored, and only the other information (the vertex dependencies) will be used.
- A subexpression is not a decoration site, regardless of what its parent is capable of. (e.g. `foo().syn` where `foo` returns a decorated type.) In this final case, the only option is to use the vertex dependencies.

In figure 5.11, we give this function $\mathscr{D}(E) : \langle VertexType?, [FlowVertex] \rangle$. If the expression has no vertex type, it will simply report *none*. We use $v.\mathit{for}(a)$ to indicate

the corresponding flow vertex for the attribute a at vertex type v. Similarly, $v.eq$ is the defining equation vertex (if any), and $v.fwd$ is the forward equation flow vertex. We occasionally use $\mathscr{D}(E)$ to refer only to the second part of this pair, but context should make it clear when this happens (see, for example, the case for function application.)

There are two other functions we use in this definition. The function $ref(N)$ is the set of inherited dependencies for taking a reference, which we introduced earlier. (We cheat on notation a little here: if E's type is not a nonterminal type, then $ref(N)$ returns the empty list.) The function $vertexType(x)$ returns the vertex type for a variable x if there is one. If the variable is the LHS, a child, a local, the forward, or references an anonymous decoration site, then it has a vertex type (as discussed previously.)

We can see that there are only two types of expressions that report having a vertex type: variables and decorate expressions. A variable may have a vertex type if it refers to the lhs, a child, a local, the forward, or pattern variable. (It has no vertex type, for example, if it refers to a production.) A decorate expression is itself an anonymous decoration site, and so always is an anonymous vertex type. We use the notation ν to refer to some internal anonymous identifier attached to the decorate expression that allows us to refer to it. Each of these shows the usual pattern for its dependencies assuming it is not used in a more precise way. Both report all the inherited attributes indicated by *ref* as dependencies, with decorate expressions additionally reporting the `eq` vertex of their anonymous vertex type. (The dependencies of E will appear later on when we start creating edges, and are not directly reported as flow vertex dependencies of the decorate expression, only indirectly through the `eq` vertex.)

Next, we see the three cases where expressions can make more efficient use of a subexpression. For `new` expressions, we need only demand `eq` and no more from the

subexpression. The obvious case, attribute access, might have a slightly non-obvious implementation: we always also include the `eq` vertex in addition to the obvious attribute vertex. For vertex types that do not have an `eq` vertex, $v.eq$ is the empty list, and so when accessing an attribute from a child (for example) only the attribute vertex is emitted as a dependency. But for vertex types that do have an `eq` vertex (locals, for example,) we must first compute the tree before it can be decorated and an attribute can be accessed from it.

The final case, pattern matching, emits direct dependencies on the equation and the forward vertex. Since pattern matching must have a tree to match against, `eq` is a dependency, and since pattern matching must be able to evaluate forward equations at the root of that tree, `fwd` is also required. This dependency on the `fwd` vertex is similar to our restriction that an extension attribute's flow types be a superset of the forward flow type. This is good, because we want the flow computations to give us the same answer for pattern matching expressions as though they had been transformed into attributes, according to our semantics for pattern matching. For most of this (just like for decorate expressions,) significant parts show up later on as part of how static production flow graphs are created, not here in the vertex dependencies of expressions. However, here we also include all vertex dependencies from each match rule (i.e. all of $\overline{E_p}$,) as these corresponds to the free variables ($\overline{x_{free}}$) in the pattern matching semantics of the previous chapter, and so will be passed to the function the attribute computes.

5.5.3 Computing intermediate flow data

Now that we have a way of getting vertexes that expressions depend upon, we turn to constructing the static production flow graphs. Recall that there are three pieces

```
   FlowDef   ::=
(hostProd)   |   Nonterminal  Production
   (extSyn)  |   Nonterminal  Attribute
(defaultEq)  |   Nonterminal  Attribute  [FlowVertex]
    (synEq)  |   Production  Attribute  [FlowVertex]  Suspect
    (inhEq)  |   Production  ChildName  Attribute  [FlowVertex]
    (fwdEq)  |   Production  [FlowVertex]  Suspect
   (fwdInh)  |   Production  Attribute  [FlowVertex]
   (localEq) |   Production  LocalName  Type  [FlowVertex]
  (localInh) |   Production  LocalName  Attribute  [FlowVertex]
   (anonEq)  |   Production  Id  Type  [FlowVertex]
  (anonInh)  |   Production  Id  Attribute  [FlowVertex]
(patternVar) |   Production  Production  VertexType  [(ChildName, Type, Id)]
```

Figure 5.12: Definition of the `FlowDef` data type as a pseudo-grammar.

we need to compute for each production: a set of edges, a set of stitch points, and a set of suspect edges. Instead of directly computing these graphs from AG trees, however, we will first compute an intermediate representation. This representation is useful because we need to use it not only for constructing static production flow graphs, but also later on to discover missing equations.

Let us begin by defining the intermediate data structure we will be using. In leiu of using some new notation, we present this definition as a grammar in figure 5.12, with the addition of giving names to each of the productions. We can immediately see two broad classes of information: per-nonterminal and per-production. We represent a set of flow vertexes in the data structure as [FlowVertex]. The components are given descriptive names: Nonterminal, Production, Attribute, ChildName, LocalName, and Id (for "anonymous" names.)

The first two per-nonterminal `FlowDef`s are intended simply as flags. For the first, if `hostProd`$(nt, prod)$ is in a set of `FlowDef`s, then *prod* is a *non-forwarding* production exported by its nonterminal. Thus, we can discover from a set of `FlowDef`s

the complete set of productions for which equations must exist (or pattern matching expressions must cover.) If `extSyn`$(nt, attr)$ is present, then *attr* is a *synthesized* attribute with an occurrence that is *not* exported by its nonterminal, and is thus an "extension attribute." This serves as an indicator of which attributes must have a flow type that is a super set of the forwarding one.

Next, we have a series of `FlowDef`s corresponding to the different kinds of equations that can appear. The `defaultEq` is unique in that it associates an attribute and a *nonterminal*, rather than a specific production, with a set of dependencies. For the rest of the equation `FlowDef`s, things are fairly straightforward: we associate a production and enough information to construct a source vertex, with the set of flow vertexes it depends upon. There are a couple subtleties, however. The `synEq` and `fwdEq` constructors are the only ones that can be marked suspect, indicating all these edges are to be introduced as suspect, rather than normal, edges. ("Suspect" in the pseudo-grammar is simply a boolean flag.) We also associate, exclusive to `localEq` and `anonEq`, the type of the nonterminal they construct. This is a bit of an implementation detail: later on when we need types, we'll have the production's signature available which gives types for the LHS, children, and forward.

The final `FlowDef` is simply an indicator of the stitch points that should be introduced as a result of pattern matching. We need several pieces of information. First, we need the target production that is being matched against, which is why Production appears twice: the first is the production this `FlowDef` (and its equation) originated from, the second is the production we'll be projecting through. Next, we need the vertex type of the scrutinee that we're matching against. Finally, we need several bits of information about the pattern variables we're extracting. The ChildName and Type are from the target production signature, and the Id is the information

needed to construct the vertex type of the pattern variable. Strictly speaking, we're wandering a bit into implementation details here. This list could just contain Ids; however, it simplifies the implementation to include the additional information, and it more easily deals with the cases where some children are ignored in the pattern using underscores.

Next, in figure 5.13 we give the computation $\mathscr{F}\,(\text{Ag}) : [\text{FlowDef}]$. To simplify the presentation, we often apply this to sets of declarations or equations, like $\mathscr{F}\left(\overline{D}\right)$, the implication being the results are appended together. Some cases are missing, where the implementation is trivial. For declarations and equations, these trivial cases produce no `FlowDef`s. For expressions, these trivial cases do nothing but descend into subexpressions.

Note that the function applies to nearly all the nonterminals of Ag: declarations and equations, of course, but also expressions, sometimes having additional context parameters for nonterminals other than declarations. For completeness, we also note the cases where orphaned declaration errors would be raised. To do so, we use the previously discussed *exports* relation, and also using *this* to indicate the module being analyzed.

$$\mathscr{F}\left(\texttt{attribute}\ \ a\texttt{<}\overline{T}\texttt{>}\ \ \texttt{occurs on}\ \ n\texttt{<}\overline{v}\texttt{>;}\right) = \begin{cases} (\text{orphaned}) & \text{if } \neg \mathit{exports}(\{n,a\}, \mathit{this}) \\ [\mathit{extSyn}(n,a)] & \text{if } a \in S \text{ and } \neg \mathit{exports}(n, \mathit{this}) \\ [] & \text{otherwise} \end{cases}$$

$$\mathscr{F}\left(\begin{array}{l}\texttt{production}\ \ x_p \\ x_l\texttt{::}n\texttt{<}\overline{T}\texttt{>::=}\ \ \overline{x_r\texttt{::}T} \\ \ \texttt{\{}\ \ \overline{Q}\ \ \texttt{\}}\end{array}\right) = \begin{cases} (\text{orphaned}) & \text{if } \neg \mathit{exports}(n, \mathit{this}) \text{ and } \mathit{fwdEq}(x_p, \_) \notin R \\ [\mathit{hostProd}(n, x_p)] +\!\!+ R & \text{if } \mathit{exports}(n, \mathit{this}) \text{ and } \mathit{fwdEq}(x_p, \_) \notin R \\ R & \text{otherwise} \end{cases}$$

$$\text{where } R = \mathscr{F}\left(\overline{Q}, x_p, n\right)$$

$$\mathscr{F}\left(\texttt{aspect}\ \ x_p\ \ \ x_l\texttt{::}n\texttt{<}\overline{T}\texttt{>::=}\ \ \overline{x_r\texttt{::}T}\ \ \texttt{\{}\ \ \overline{Q}\ \ \texttt{\}}\right) = \mathscr{F}\left(\overline{Q}, x_p, n\right)$$

$$\mathscr{F}\left(\texttt{default}\ \ \ \ x_l\texttt{::}n\texttt{<}\overline{v}\texttt{>::=}\ \ \texttt{\{}\ \ \overline{Q}\ \ \texttt{\}}\right) = \mathscr{F}\left(\overline{Q}, \mathit{default}, n\right)$$

$$\mathscr{F}\left(x\ \texttt{.}\ a\ \ \texttt{=}\ \ E\texttt{;}, p, n\right) = \begin{cases} (\text{orphaned}) & \text{if } \neg \mathit{exports}(\{p,o\}, \mathit{this}) \text{ where } o \text{ is occurs for } a \text{ on } x\text{'s type} \\ [\mathit{synEq}(p, a, \mathscr{D}(E), \mathit{exports}(\{n,o\}, \mathit{this}))] +\!\!+ \mathscr{F}\left(E, p\right) & \text{if } a \in S \text{ and } x \text{ is LHS} \\ [\mathit{inhEq}(p, x, a, \mathscr{D}(E))] +\!\!+ \mathscr{F}\left(E, p\right) & \text{if } a \in I \text{ and } x \text{ is RHS} \\ [\mathit{defaultEq}(n, a, \mathscr{D}(E))] +\!\!+ \mathscr{F}\left(E, p\right) & \text{if } p = \mathit{default} \\ [\mathit{localInh}(p, x, a, \mathscr{D}(E))] +\!\!+ \mathscr{F}\left(E, p\right) & \text{if } a \in I \text{ and } x \text{ is local} \end{cases}$$

$$\mathscr{F}\left(\texttt{forwards to}\ \ E\ \ \texttt{\{}\ \ \overline{A}\ \ \texttt{\};}, p, n\right) = [\mathit{fwdEq}(p, \mathscr{D}(E), \mathit{exports}(n, \mathit{this}))] +\!\!+ \mathscr{F}\left(E, p\right) +\!\!+ \mathscr{F}\left(\overline{A}, p, \mathit{fwd}\right)$$

$$\mathscr{F}\left(\texttt{local}\ \ x\texttt{::}T\ \ \texttt{=}\ \ E\texttt{;}, p, n\right) = [\mathit{localEq}(p, x, T, \mathscr{D}(E))] +\!\!+ \mathscr{F}\left(E, p\right)$$

$$\mathscr{F}\left(\texttt{decorate}\ \ E\ \ \texttt{with}\ \ \texttt{\{}\ \ \overline{A}\ \ \texttt{\}}, p\right) = [\mathit{anonEq}(p, \nu, T, \mathscr{D}(E))] +\!\!+ \mathscr{F}\left(E, p\right) +\!\!+ \mathscr{F}\left(\overline{A}, p, \mathit{anon}(\nu)\right) \text{ where } E :: T$$

$$\mathscr{F}\left(\texttt{case}\ \ E\ \ \texttt{of}\ \ \overline{P\ \ \texttt{->}\ \ E_p}, p\right) = R +\!\!+ \mathscr{F}\left(E, p\right) +\!\!+ \mathscr{F}\left(\overline{E_p}, p\right)$$

$$\text{where } R = \begin{cases} \mathscr{F}\left(\overline{P}, p, v\right) & \text{if } \mathscr{D}(E) = \langle v, \_ \rangle \\ \mathscr{F}\left(\overline{P}, p, \nu\right) +\!\!+ [\mathit{anonEq}(p, \nu, T, [])] & \text{otherwise (where } E :: \texttt{Decorated}\,T) \end{cases}$$

$$\mathscr{F}\left(x_p(\overline{x}), p, v_s\right) = [\mathit{patternVar}(p, x_p, v_s, \overline{x :: T \mapsto \nu})]$$

$$\mathscr{F}\left(a\ \ \texttt{=}\ \ E, p, ty\right) = \mathscr{F}\left(E, p\right) +\!\!+ \begin{cases} [\mathit{fwdInh}(p, a, \mathscr{D}(E))] & \text{if } ty = \mathit{fwd} \\ [\mathit{anonInh}(p, x, a, \mathscr{D}(E))] & \text{if } ty = \mathit{anon}(x) \end{cases}$$

Figure 5.13: Computing flow definitions from AG. Trivial repetitive cases omitted.

For declarations (D), we can see all cases are fairly simple. For attribute occurrences, we may find an orphaned occurrence, or report an occurrence as being part of an "extension" (and thus have a minimum flow type.) For original production declarations, we may report it as orphaned or as being a legally non-forwarding "host language" production. We otherwise simply descend into the equations.

When checking equations (Q), we include some extra context, which is the production and LHS nonterminal for that production. (For default productions, we simply report *default* as a placeholder for the production. We once again see the possibility of orphaned equations, and note that we use o to refer to the module the attribute occurrence appears in. Beyond that, we just determine exactly what kind of equation it is, and report the correct `FlowDef`. Here, we don't directly handle certain already erroneous cases. For example, we would have already raised errors for trying to define an inherited attribute for the LHS, or a forward equation in an aspect production.

Notice, for `synEq` and `fwdEq`, we finally see our determination for whether these equations should be considered suspect. These rules follow the same logic as orphaned declarations or equations. If a synthesized equation appears in a module exported by the nonterminal or the attribute occurrence, then it is non-suspect. Anyone accessing that attribute must have imported the occurrence and that nonterminal's module, and so must be aware of this equation's influence on flow types. Likewise, a forwarding equation is only non-suspect if it is exported by the production's nonterminal.

Next, we see the only two cases for expressions that contribute `FlowDef`s (followed by two cases for syntax within these expressions.) As extra context, expressions need only know what production they appear within. Both these expressions potentially make use of anonymous identifiers (once again denoted ν) associated with the node in the tree. Decorate expressions always do so, providing this id to the `anonEq` and

down to the inherited equations list as well. Pattern matching expressions will use the scrutinee's vertex type, unless there isn't one, in which case a new anonymous vertex type is created as a proxy. The purpose of this is to simplify how we handle pattern variables, since we can now always assume there is a vertex type for the scrutinee. We do not indicate any dependencies for this new vertex, it's purpose is only to accumulate dependencies for our later check for completeness.

For patterns, we include not only the production we're within, but also the vertex type of the scrutinee (v_s) as extra context. When reporting the projection stitch point information, the anonymous identifier (ν) is associated with each pattern variable, not the pattern as a whole (that is, each pattern variable has its own identity.) For inherited equations, we include as context, in addition to the production, what we're supply inherited equations to. This syntax can appear within decorate expressions or the forwarding equation, and so we emit two different kinds of equations, depending.

For the expressions not listed, this function simply proceeds to accumulate the `FlowDef`s from its subexpressions.

5.5.4 Constructing static production flow graphs

We can now give a function from this intermediate flow information to static production flow graphs. This function (and associated utility functions) are defined in figure 5.14. Its inputs are the *flow environment* (the concatenation of all FlowDefs from all modules transitively depended upon) and the typing environment (likewise concatenated over all modules.) Excepting the utility functions, the basic structure of this function is relatively simple. We turn the flow information for each production into a graph, with behavior that depends on whether the production is forwarding or not. If the production does not forward, then we introduce implicit edges that are a

result of a default equations via the function *defaultEdges*.

For forwarding productions, we introduce edges for implicit equations in two parts. For implicit inherited equations, we introduce the edges via the function *fwdInhs*. This adds copies of all inherited attributes from the LHS to the forward node, if there is no such explicit equation given. For implicit synthesized equations, we add all edges via the function *fwdSyns*, but we treat all these edges as suspect. These edges may be for synthesized attributes that the forwarding production has no business affecting. (One possible design decision would be to decide which of these edges might be considered non-suspect on the basis of what attributes are known to exist in the grammar declaring the forwarding production, however this is more complicated and does not necessarily improve the analysis.)

The edges for explicit equations are introduced into the production graph by the functions *edges* and *suspectEdges* for normal edges and suspect ones respectively. Note that these functions do non-linear matching: we're essentially filtering down to just those pieces of flow information specific to the production P we're constructing a flow graph for. These are a mostly straight-forward translation of the intermediate data structure. We write $edge(s, \bar{t})$ to represent creating many edges from the source s to the list of targets t. The cases handling synthesized and forward equations are identical between *edges* and *suspectEdges*, except whether the equations are flagged as being permitted to affect the flow type. That is, we're just filtering them differently.

For $defaultEdges$, $fwdInhs$, and $fwdSyns$, we introduce an edge only if there does not already exist an explicit edge, thus why we also pass in F (the complete flow environment.) Notice, once again, that the implicit copy equations for the forward synthesized attributes also depend on the `eq` vertex of the forward, just as synthesized attribute access does.

Given $F :: [\mathit{FlowDef}], E_{N,P,S,I,O}$ (typing environment information for all modules)
For each production $P :: (LHS ::= RHS_i) \in E_P$
Yield $(LHS, \mathit{Graph}, \mathit{Stitch}, \mathit{Suspects})$ where
$\mathit{fwding} = (\mathit{fwdEq}(P, \_) \in F)$
$\mathit{syns} = [s \text{ for } s \in E_S \text{ where } s@LHS \in E_O]$
$\mathit{inhs} = [i \text{ for } i \in E_I \text{ where } i@LHS \in E_O]$
$\mathit{fwdingEdges} = \mathit{fwdInhs}(\overline{\mathit{inhs}}, F, P)$
$\mathit{nonfwdingEdges} = \mathit{defaultEdges}(\overline{F}, F, P, LHS)$
$\mathit{Graph} = \mathit{edges}(\overline{F}, P) +\!\!+ (\text{if } \mathit{fwding} \text{ then } \mathit{fwdingEdges} \text{ else } \mathit{nonfwdingEdges})$
$\mathit{Stitch} = \mathit{childStitches}(\overline{RHS_i}) +\!\!+ \mathit{flowStitches}(\overline{F}, P)$
$\mathit{Suspects} = \mathit{suspectEdges}(\overline{F}, P) +\!\!+ (\text{if } \mathit{fwding} \text{ then } \mathit{fwdSyns}(\overline{\mathit{syns}}, F, P) \text{ else } [])$

$\mathit{edges}, \mathit{suspectEdges} :: \mathit{FlowDef} \to \mathit{Production} \to [\mathit{FlowEdge}]$
$\mathit{edges}(\mathit{synEq}(P, A, \overline{D}, \mathit{false}), P) = [\mathit{edge}(\mathit{lhsVertex}.\mathit{for}(A), \overline{D})]$
$\mathit{edges}(\mathit{inhEq}(P, R, A, \overline{D}), P) = [\mathit{edge}(\mathit{rhsVertex}(R).\mathit{for}(A), \overline{D})]$
$\mathit{edges}(\mathit{fwdEq}(P, \overline{D}, \mathit{false}), P) = [\mathit{edge}(\mathit{fwdVertex}.\mathit{eq}, \overline{D})]$
$\mathit{edges}(\mathit{fwdInh}(P, A, \overline{D}), P) = [\mathit{edge}(\mathit{fwdVertex}.\mathit{for}(A), \overline{D})]$
$\mathit{edges}(\mathit{localEq}(P, X, T, \overline{D}), P) = [\mathit{edge}(\mathit{localVertex}(X).\mathit{eq}, \overline{D})]$
$\mathit{edges}(\mathit{localInh}(P, X, A, \overline{D}), P) = [\mathit{edge}(\mathit{localVertex}(X).\mathit{for}(A), \overline{D})]$
$\mathit{edges}(\mathit{anonEq}(P, X, T, \overline{D}), P) = [\mathit{edge}(\mathit{anonVertex}(X).\mathit{eq}, \overline{D})]$
$\mathit{edges}(\mathit{anonInh}(P, X, A, \overline{D}), P) = [\mathit{edge}(\mathit{anonVertex}(X).\mathit{for}(A), \overline{D})]$

$\mathit{suspectEdges}(\mathit{synEq}(P, A, \overline{D}, \mathit{true}), P) = [\mathit{edge}(\mathit{lhsVertex}.\mathit{for}(A), \overline{D})]$
$\mathit{suspectEdges}(\mathit{fwdEq}(P, \overline{D}, \mathit{true}), P) = [\mathit{edge}(\mathit{fwdVertex}.\mathit{eq}, \overline{D})]$

$\mathit{defaultEdges} :: \mathit{FlowDef} \to [\mathit{FlowDef}] \to \mathit{Production} \to LHS \to [\mathit{FlowEdge}]$
$\mathit{defaultEdges}(\mathit{defaultEq}(LHS, A, \overline{D}), F, P, LHS) =$
$\quad \text{if } \mathit{synEq}(P, A, \_, \_) \in F \text{ then } [] \text{ else } [\mathit{edge}(\mathit{lhsVertex}.\mathit{for}(A), \overline{D})]$

$\mathit{fwdSyns}, \mathit{fwdInhs} :: \mathit{Attribute} \to [\mathit{FlowDef}] \to \mathit{Production} \to [\mathit{FlowEdge}]$
$\mathit{fwdSyns}(A, F, P) = \text{if } \mathit{synEq}(P, A, \_, \_) \in F \text{ then } [] \text{ else}$
$\quad [\mathit{edge}(\mathit{lhsVertex}.\mathit{for}(A), \mathit{fwdVertex}.\mathit{for}(A)), \mathit{edge}(\mathit{lhsVertex}.\mathit{for}(A), \mathit{fwdVertex}.\mathit{eq})]$

$\mathit{fwdInhs}(A, F, P) = \text{if } \mathit{fwdInh}(P, A, \_) \in F \text{ then } [] \text{ else}$
$\quad [\mathit{edge}(\mathit{fwdVertex}.\mathit{for}(A), \mathit{lhsVertex}.\mathit{for}(A))]$

$\mathit{childStitches} :: \mathit{RhsElement} \to [\mathit{StitchPoint}]$
$\mathit{childStitches}(x :: T) = [\mathit{ntStitchPoint}(\mathit{rhsVertex}(x), T)]$

$\mathit{flowStitches} :: \mathit{FlowDef} \to \mathit{Production} \to LHS \to [\mathit{StitchPoint}]$
$\mathit{flowStitches}(\mathit{fwdEq}(P, \_, \_), P, LHS) = [\mathit{ntStitchPoint}(\mathit{localVertex}(X), LHS)]$
$\mathit{flowStitches}(\mathit{localEq}(P, X, T, \_), P, LHS) = [\mathit{ntStitchPoint}(\mathit{localVertex}(X), T)]$
$\mathit{flowStitches}(\mathit{anonEq}(P, X, T, \_), P, LHS) = [\mathit{ntStitchPoint}(\mathit{anonVertex}(X), T)]$
$\mathit{flowStitches}(\mathit{patternVar}(P, P_T, V_S, \overline{(X, T_X, V_X)}), P, LHS) =$
$\quad [\mathit{ntStitchPoint}(\mathit{anonVertex}(X), T_X)] +\!\!+ [\mathit{projectionStitchPoint}(P_T, V_X, V_S, X)]$

Figure 5.14: Computing production flow graphs.

Given $F :: [FlowDef], E_{N,P,S,I,O}$ (typing environment information for all modules)
For each nonterminal $N \in E_N$
Yield $(N, Graph, Stitch, Suspect)$ where
$Graph = phantomEdges(\overline{F}, N)$
$Stitch = [ntStitchPoint(lhsVertex, N)]$
$Suspect = []$

$phantomEdges :: FlowDef \to LHS \to [FlowEdge]$
$phantomEdges(extSyn(N, A), N) = [edge(lhsVertex.for(A), lhsVertex.fwd)]$

Figure 5.15: Computing phantom production flow graphs.

Finally, stitch points are introduced in two categories: child stitch points resulting from the production signature, and those stitch points that result from other information in the flow environment. These include forwarding, anonymous decoration sites, locals, and pattern variables. Introducing these stitch points is why we passed along type information into the `FlowDef`s earlier, otherwise we would not know which nonterminal's flow type is being stitched in here. Pattern variables are special in that they induce both nonterminal stitch points and projection stitch points.

This is sufficient to construct static production flow graphs. However, there is one more matter to attend to before we start making use of these graphs. We must handle those synthesized attributes which are introduced by extensions, and we must enforce the property that $ft_{nt}(syn_{ext}) \supseteq ft_{nt}(fwd)$. Instead of (unhelpfully) raising errors whenever this is not the case, we instead just ensure that extension synthesized attribute always compute flow types with the forward flow type as a lower bound. We accomplish this by introducing phantom production flow graphs that do not correspond to any particular production. Figure 5.15 shows how we compute these productions.

Phantom production flow graphs have a couple of notable differences from or-

dinary productions. They introduce a nonterminal stitch point for the LHS, where normal production flow graphs do not. There are no suspect edges. And finally, the only edges they directly contain are from extension synthesized attributes to the forwarding equation. This has the effect of ensuring that every extension synthesized attribute depends on at least those inherited attributes required by forwarding.

5.5.5 Computing flow types

At this point, having computed the static production flow graphs, we can now apply a slight variation on the usual attribute grammar flow computation. This variation is motivated by three essential differences:

1. We have some edges we consider suspect, and must treat specially.
2. We have another kind of stitch point (projection) beyond the standard one (nonterminal.)
3. We added a phantom production for each nonterminal as well.
4. We don't need to compute flow sets, we care only about flow types.

When we introduced flow types, we already described how they can be computed more efficiently and directly than flow sets. This means instead of tracking a set of flow graphs per nonterminal, we track just one flow type per nonterminal. Instead of discovering new flows and adding them to the set, we just update the flow type to include any new potential dependencies. To provide more details about the other changes, we first present the algorithm.

The flow analysis is a fixed point algorithm across all productions of all nonterminals (now also including our phantom productions.) Each time a production is

visited, it will update itself and the flow type of its LHS nonterminal. The initial state for flow types is empty: we assume no synthesized attribute depends on any inherited attribute. The initial state for each production is the static production flow graphs we have computed.

When a production graph is visited by the algorithm, we begin by updating every stitch point in the current production graph. Nonterminal stitch points may introduce new internal edges within a vertex type according to their corresponding flow type, and projection stitch points introduce new edges for inherited attributes between two vertex types according to another production's current flow graph. Once these stitch points are updated, we compute the transitive closure on edges, the result is called a *stitched flow graph*. Any new paths in the production graph from tracked vertexes (LHS synthesized attributes or the forward equation) to LHS inherited attributes are added as new edges in the LHS nonterminal's flow type. This procedure repeats over every production graph until no new edges are introduced to any graphs. (Our termination condition considers all graphs, not just flow types, because of projection stitch points, which may cause a change in one production .)

However, we have not yet described how to handle suspect edges. It turns out we cannot ever directly introduce suspect edges to the regular production flow graph. Consider this example:

```
production extension
e::HostNT ::=
{
  e.hostSyn1 = ... e.inh ...;
  e.hostSyn2 = ... e.hostSyn1 ...;
  forwards to ...
}
```

In this example, both edges are considered suspect. However, suppose we *first* decide to admit the `hostSyn2` → `hostSyn1` suspect edge as an ordinary edge, since we do not yet observe any ordinary edges indicating this would violate `hostSyn2`'s flow type. We might then *later* decide to admit the `hostSyn1` → `inh` edge, as perhaps this does not violate `hostSyn1`'s flow type. However, we've now introduced a transitive dependency from `hostSyn2` to `inh` which *may* violate the flow type of `hostSyn2`. Because of these transitive effects, we cannot ever directly turn a suspect edge into an ordinary one.

Instead, we introduce only **direct** edges from the source of the suspect edge to the resulting inherited attribute on the LHS. That is, we skip over the target of the suspect edge, and consider only the potential dependencies it might induce as flow types (that is, LHS inherited attributes.) Production flow graphs will never have edges *from* an inherited attribute on the LHS, and so we've eliminated the transitivity problem. We'll never get more dependencies than those we explicitly introduce, having checked against the flow type that they are okay.

As a result, we never directly introduce `hostSyn2` → `hostSyn1` as an ordinary edge. We would accept `hostSyn1` → `inh` only if the flow type for `hostSyn1` allows it. And since our scenario requires `hostSyn2` to have no dependency on `inh`, when we consider the `hostSyn2` → `hostSyn1` suspect edge, we would only consider adding the `hostSyn2` → `inh` direct edge, and reject it. (The task of raising an error, because this equation depends upon something it is not allowed to, is the subject of the next section.)

5.6 Checking for effective completeness

At this point, we have ensured that there are no orphaned productions, and we have computed flow types that respect the modular analysis's restrictions. Now we must analyze the grammar, to determine whether there are any other completeness problems.

The first part of checking effective completeness is quite easy. To ensure all synthesized equations are present, we simply check for their presence from the synthesized attribute occurrence. That is, the occurrence of at on nt should find:

$$\{p|hostProd(nt, p) \in F\} \setminus \{p|synEq(p, at, \_, \_) \in F\} = \emptyset$$

where F is the flow environment consisting of only the module's dependencies. Because the occurrence must import the nonterminal module, and that module must export all non-forwarding productions, this is sufficient to ensure all synthesized equations are present.

To improve error messages, it might be helpful to divide this task into two. If the synthesized attribute occurrence is also exported by the nonterminal's module, then instead of checking for missing equations on the occurrence, we can do so on the production declaration instead. This has the benefit of raising the error message in the location that a fix should be made to resolve it, instead of wherever the occurrence appears.

Next, we can ensure there are no duplicate equations of any kind, synthesized or inherited. Since each kind of equation gives exactly one kind of `FlowDef`, we can have every equation check to ensure there is only one of its kind in F. Thanks once again to the orphan restrictions, we can always be sure that if such a duplicate exists, it will appear there.

The task of ensuring all necessary inherited equations are present is more involved. We must check to ensure certain equations are present, but we must also check to ensure we do not exceed our allowed dependencies, and the latter is the more intricate task.

5.6.1 Effective inherited completeness

In this section, we will present an analysis that performs two essential tasks. First, it will check for the presence of certain equations. Second, it will check that equations not depend on things they are not allowed to.

This second part comes in two general forms. First, we need to ensure each equation does not exceed its corresponding flow type (if it has one.) This is essential for the prior inherited equation presence check to be effective. Second, we need to handle the subtleties of some language features in Ag. These include default equations, pattern matching, reference attributes, and forwarding. Each of these has some special restrictions to enforce.

In figure 5.16, we present a set of functions that performs this final task. The *check* function takes an attribute grammar and decomposes the task down to its parts. We write *check* as a boolean function that returns true if everything is fine, and false otherwise (and thus some suitable error should be raised.) To begin with the easy parts, the *depsCheck* function ensures all necessary inherited equations exist, and the *exceedsCheck* function ensures that a suspect equation does not exceed its corresponding flow type.

We often apply *depsCheck* to a set of flow vertexes, this simply means applying it to each flow vertex. The *depsCheck* function operates on a flow vertex within a production, and (if it is an inherited equation for a decoration site,) we check to

ensure that the equation exists. This function essentially translates vertexes into the necessary inherited equations that need to exist. Specifically, checking for the kind of FlowDef from figure 5.12 that corresponds to that type of inherited equation (so *inhEq* for inherited equations given to a child, and so on.) Notably absent are inherited vertexes for forward vertex type, as these are always present (either as explicit equations or generated implicitly as copies.)

The next major tool we use is the *exceedsCheck* function, which takes a set of dependencies, the production graph to look within, and the flow type that these dependencies must be bounded by. This function transitively closes the set of dependencies using the production graph (using $edgesFrom$), filters this down to inherited attributes on the LHS ($onlyLhsInh$), and then ensures that thus is a subset of the input bound.

Returning to the *check* function, we see that *depsCheck* is applied for each equation, and *exceedsCheck* is applied for each item that has a tracked flow type (synthesized attribute equations and the forward equation.) These constitute the main purpose of this check, and now all that remains are special cases for interesting language features.

For defaults, we do not have a production (or production graph) to use to perform the usual checks. However, we accomplish this in a straightforward way: since we know each production the default applies to, we simply perform those checks on each production that default equation would have been copied to. Note that we do not need to perform the exceeds check for default equations. Since these equations must be exported by their occurrence already, they are never suspect and do not need the exceeds check.

For forwarding, we have a couple of special checks accomplished by $fwdInhCheck$

Given a flow environment of transitive dependencies F, a flow type solution ft, a set of final production graphs G indexed by name ($G(P)$ is the graph for production P,) and the usual typing environment $E_{N,P,S,I,O}$:

$edgesFrom, edgesTo :: FlowVertex \to ProductionGraph \to [FlowVertex]$
$onlyLhsInh, onlyInh :: [FlowVertex] \to [Attribute]$

$check :: \textsc{Ag} \to Error$
$check(\texttt{production}\ \ x_p\ \ x_l\ \texttt{::}\ n_l\ \ldots\ \{\overline{Q}\}) = check(\overline{Q}, x_p, n_l)$
$check(\texttt{aspect}\ \ x_p\ \ x_l\ \texttt{::}\ n_l\ \ldots\ \{\overline{Q}\}) = check(\overline{Q}, x_p, n_l)$
$check(\texttt{default}\ \ x_l\ \texttt{::}\ n_l\ \{\overline{Q}\}) = defaultCheck(\overline{Q}, n_l)$

$check(\texttt{local}\ \ x\ \texttt{::}\ T\ \ \texttt{=}\ \ E\ \texttt{;}\ , P, N) = check(E, P, N) \wedge depsCheck(\overline{\mathscr{D}(E)}, P)$
$check(x\ \texttt{.}\ a\ \ \texttt{=}\ \ E\ \texttt{;}\ , P, N) = check(E, P, N) \wedge depsCheck(\overline{\mathscr{D}(E)}, P) \wedge$
$\quad \big(a \in E_S \implies exceedsCheck(\mathscr{D}(E), G(P), ft_N(a))\big)$
$check(\texttt{forwards to}\ \ E\ \ \texttt{\{}\ \ \overline{A}\ \ \texttt{\}}\ \texttt{;}\ , P, N) = check(E, P, N) \wedge depsCheck(\overline{\mathscr{D}(E)}, P) \wedge$
$\quad exceedsCheck(\mathscr{D}(E), G(P), ft_N(fwd)) \wedge fwdInhCheck(\overline{A}, P, N) \wedge$
$\quad implicitFwdCheck(P, N)$

$check(E\ \texttt{.}\ a, P, N) = check(E, P, N) \wedge$
$\quad \big(\mathscr{D}(E) = \langle none, \_\rangle \wedge E :: \texttt{Decorated}\, n \implies (ft_n(a) \setminus ref(n) = \emptyset)\big)$
$check(\texttt{case}\ \ E\ \ \texttt{of}\ \ \overline{M\ \ \texttt{->}\ \ E_p}, P, N) = check(\{E\} \cup \overline{M} \cup \overline{E_p}, P, N) \wedge$
$\quad \big(\mathscr{D}(E) = \langle none, \_\rangle \wedge E :: \texttt{Decorated}\, n \implies onlyInh(edgesTo(\nu, G(P))) \setminus ref(n) = \emptyset\big)$

$check(x_p(\overline{x}), P, N) = depsCheck(edgesTo(\overline{\nu}, G(P))[\nu \mapsto rhs], x_p)$

$depsCheck :: FlowVertex \to Production \to Error$
$depsCheck(rhsVertex(X, A), P) = A \in E_I \implies inhEq(P, X, A, \_) \in F$
$depsCheck(localVertex(X, A), P) = A \in E_I \implies localInhEq(P, X, A, \_) \in F$
$depsCheck(anonVertex(X, A), P) = A \in E_I \implies anonInhEq(P, X, A, \_) \in F$

$exceedsCheck :: [FlowVertex] \to ProductionGraph \to [Attribute] \to Error$
$exceedsCheck(V, G, A) = onlyLhsInh(edgesFrom(\overline{V}, G)) \setminus A = \emptyset$

$defaultCheck :: Q \to Nonterminal \to Error$
$defaultCheck(x\ \texttt{.}\ a\ \ \texttt{=}\ \ E\ \texttt{;}\ , N) = \forall P.\ hostProd(N, P) \in F \wedge synEq(P, a, \_) \notin F \implies$
$\quad check(E, P, N) \wedge depsCheck(\overline{\mathscr{D}(E)}, P)$

$fwdInhCheck :: A \to Production \to Nonterminal \to Error$
$fwdInhCheck(a\ \ \texttt{=}\ \ E, P, N) = exceedsCheck(\mathscr{D}(E), G(P), ft_N(fwd) \cup \{a\})$

$implicitFwdCheck :: Production \to Nonterminal \to Error$
$implicitFwdCheck(P, N) = \forall s \in E_S,$
$\quad s@N \in E_O \wedge extSyn(N, s) \notin F \wedge synEq(P, s, \_) \notin F \implies ft_N(s) \supseteq ft_N(fwd)$

Figure 5.16: Checking for missing inherited equations and disallowed dependencies.

and *implicitFwdCheck*. We introduced as suspect edges all of the implicit synthesized copy equations for forwarding. In the general case, this is fine: extension-introduced synthesized attributes must have a flow type that is a superset of the forwarding flow type thanks to phantom productions. However, non-extension synthesized attributes may have a flow type which is less than the forwarding one. For instance, we may have a "pretty printing" attribute that requires no inherited attributes, while the forwarding flow type requires an environment. This would be a problem if this attribute is computed via forwarding. This means the host language has created a burden on all productions to provide an explicit, no-dependencies equation for this "pretty-printing" attribute. We enforce this by checking, for every production declaration, for every host language attribute (which we know about because they are exported by the nonterminal declaration,) to ensure that the implicit equation does not exceed the flow type. This is accomplished by the *implicitFwdCheck* function.

The $fwdInhCheck$ function handles an unusual special case that forwarding brings up. Normally, inherited equations have no direct restrictions, and are limited only in the sense that other equations may depend on them (and thus we might have multiple, indirect limitations.) However, forwarding is unusual because new copy equations might be introduced that appear in no module. This affect inherited equations indirectly via those synthesized copy equations, which are never checked. Thus, the inherited equations given down to the forward tree must be conservative in their dependencies: we can rely only on the forwarding flow type (which we must have available if we're decorating the forward tree) and the particular attribute being defined. This is accomplished by $fwdInhCheck$.

For references, we have two new special cases to handle, both in areas where we

might use a reference (a decorated value) without an associated decoration site. That is, any place we obtain a decorated value from elsewhere, rather than decorating it locally (reference attributes, for example.) The only places where we emit dependencies are for the attribute access and pattern matching expressions (skipping `new` because it doesn't directly emit any attribute dependencies.) And so, when *check* descends into expressions, we check to ensure these accesses on references do not exceed *ref*(n) where n is the nonterminal type of the subexpression. Thus, by emitting dependencies on the *ref* set, we have ensured that all such inherited attributes are supplied, and here we restrict the allowed dependencies to just those in the *ref* set, completing the circle for references.

Finally, we have pattern matching to handle. This special case is handled by descending into patterns themselves in the *check* function. The case for patterns emits (once again calling *depsCheck*) a special check *on another production* (the one the pattern matches.) Recall that our translation of pattern matching into attributes would emit equations for other productions, thus the reasons for this sort of remote equation check.

This check works by first finding all edges to each of the anonymous decoration sites we created for each pattern variable (the vertex type for x denoted ν.) We then translate those dependencies from the anonymous vertex type in this production to the child vertex type in the remote production. To indicate this translation to child vertex types, we abuse notation by writing $[\nu \mapsto rhs]$ to indicate this translation. Finally, we perform the remote *depsCheck* for the existence of those vertexes. For example, consider the projection stitch point example in figure 5.10. If we have a dependency on '`x.I`', then we need to map this to '`rhs.I`' and then run the *depsCheck* on the production `foo`. This ensures that `foo` has an inherited equation for '`I`' on

that child.

5.6.2 Implementation notes

This presentation of the effective completeness analysis is complete, but there are a few additional notes that should be taken regarding its implementation. First, we determine when there are missing inherited equations on an equation-by-equation basis, however it may be helpful to be more specific about the reasons why. For instance, we can introduce a new check on attribute accesses that also checks for missing equations. This way the error message can be refined from "this equation depends on an inherited attribute 'inh' of this child, but no equation is given" to "this access of 'child.syn' requires an equation defining 'child.inh'." This can help users discover where dependencies are coming from.

We have also chosen to check for missing equations provided to anonymous decoration sites within *depsCheck*, however this may lead to confusing errors, as the decoration site has no real name to refer to. We could also look for *edgesTo* the anonymous decoration site, and then locally raise errors from the `decorate` expression itself. This way, the user gets errors from both the location where equations must be added, as well as the location causing the dependency. Debugging the code may go either way: adding the equation, or altering another expression to no longer (erroneously) depend upon that inherited attribute, and there is no clear way to see which is intended.

Next, we must be careful about changes to language features that could impact our assumptions about implicit equations. For default equations in Ag, the implicit equations are always non-suspect and do not need checking for exceeding flow types. However, if we permitted flow types to be declared along with the attribute occurrence

declaration (i.e. specify the flow type, instead of inferring it,) then default equations might exceed that stated flow type, and currently there is no check for this.

5.6.3 Extending to additional language features

Silver proper (as opposed to AG) does include a few additional features that have implications for this analysis. One of these is "collection attributes" to which extensions can contribute values in a well-defined way. The particulars of this feature are not important here, but it is worth noting that this means extensions can introduce expressions that would potentially influence the dependencies of a "base" equation written in the host language. We can accommodate these by simply requiring that the extension's "contributions" to the equation's value cannot depend upon any more inherited attributes that the original base equation does.

Silver also possesses a notion of a "closed nonterminal." These are useful for describing concrete syntax, where we wish to take advantage of a dual form of extensibility. Normal nonterminals forbid new (non-forwarding) productions, but allow any new attributes. This is the most useful form for abstract syntax, but for concrete syntax trees this is questionable. We would instead like to permit new productions, but forbid new attributes. There is surprisingly little effect on the effective completeness analysis, except in terms of removing the orphaned production check (for these "closed nonterminals") and introducing such a check on attribute occurrences. Each synthesized attribute occurrence must be exported by the closed nonterminal, or it must have a default equation giving its value in terms of other attributes on the nonterminal. All flow analysis concerns work identically, complete with the standard rules for when equations are permitted to affect the flow type.

Finally, Silver also introduces "autocopy attributes" which automatically copy

their value, unchanged, from the parent production to all children. This works a lot like how inherited attributes are copied implicitly to the forward decoration site. The changes to the effective completeness analysis are modest: we simply need to introduce edges appropriately when constructing the static production flow graphs, and we need to amend the error check for $rhsVertex$ in $depsCheck$ to also ensure that the missing inherited equation is not for an autocopy attribute that also occurs on the LHS.

More interestingly, however, is that these autocopy equations are not statically determinable in a modular way. To avoid issuing duplicate equations, we require the equation declarations to be non-orphan, that is, exported by the occurrence or the production declaration. But it is easy to write a (forwarding) production declaration that is totally unaware of an autocopy attribute, and likewise an attribute occurrence unaware of that production. Dealing with autocopy equations then is much like dealing with implicit synthesized equations as a result of forwarding, where new equations may appear as a result of introducing new modules. However, while we needed special rules to ensure these implicit forwarding synthesized equations do not exceed the corresponding flow type, we have no such concerns for inherited attributes. And so, while autocopy attributes do not pose a problem for the analysis, they are unfortunate in that autocopy equations must be generated at composition time in order to avoid introducing duplicate equations.

Finally, we may also wish to permit more complex default equations. For example, we might wish to support "functor attributes" or "monoid attributes" which yield implicit equations for synthesized attributes that will recursively make use of children on which the same attribute occurs. These are fundamentally identical to default equations, except that the particular implicit equation they have will vary

per-production, rather than per-nonterminal. And so, like default equations, they will need special treatment in how static production flow graphs are created, and (if translated away) how equations are inserted statically.

5.7 Self-evaluation on Silver

This analysis places additional restrictions on the relationships between modules, especially where host language and extension are concerned. Later on, in chapter 7, we construct a C compiler and some extensions that all satisfy this analysis. As evaluations of whether useful extensions are possible for practical host languages, this should be adequate. To evaluate this analysis on an existing attribute grammar, though, we have applied it to the Silver compiler itself, which is written in Silver.

The Silver host language is one of the most complex Silver programs we have. It was implemented in several modules, a division that we found natural prior to the development of this analysis. It has several language features that are implemented not in the "host language" proper, but as extensions, and so should be subject to the restrictions of this analysis. As a further benefit, it may be interesting to note where we do not already meet the requirements of the analysis, given that Silver was largely developed prior to this analysis. It is also the Silver specification that we use the most, and as a result we believe it would have the fewest bugs.

We briefly describe a few of the extensions to the Silver compiler, to demonstrate they are interesting and nontrivial. A "convenience" extension introduces new syntax that greatly simplifies making large numbers of similar occurs declarations, by allowing nonterminal declarations to be annotated with a list of them (this is source of our `with` syntax on nonterminals.) A "testing" extension adds several constructs for

writing and generating unit tests for the language specification. An "easy terminals" extension allows keyword terminals to be referred to by their lexeme in production signatures instead of using the name the terminal was declared under (using `'to'` instead of `To_kwd`, for example.) We also separate primitive pattern matching (part of the host language, as described for AG) and a language extension that permits nested pattern matching on multiple values. Finally, the entire translation to Java is implemented as a composable language extension.

A technical report documents the details of the issues raised by the analysis and the changes made to address them [80]. We discuss the interesting aspects that arose from this exercise here.

Silver focuses specifically on language extension, and as a result, we had not previously implemented the monolithic, whole-program well-definedness analysis for attribute grammars. Without the modular analysis, we had simply gotten by without a static completeness analysis at all, relying on dynamic errors instead. The first set of changes to satisfy the analysis were simple bug fixes that could have been caught with a whole-program analysis. We found several legitimately missing synthesized and inherited attribute equations. It also found several productions that should have been forwarding, but were not. These were not exposed because we had written equations for all synthesized attributes that were ever actually demanded. In other words, these were lurking bugs, unexposed until we would have tried to write a certain type of entirely valid language extension.

Another positive set of changes improved the quality of the implementation, even if they did not directly fix latent crash-bugs. We discovered several extraneous attribute occurrences that simply never had equations, and were never used either. Many uses of reference attributes were found to be completely unnecessary and removed.

One particularly interesting change has to do with how concrete syntax specifications are handled in Silver. Silver's host language supplies a "standard" set of declarations for concrete syntax (nonterminals, terminals, etc), while Copper-specific declarations are kept in a separate module. The analysis raised a simple error: the Java translation attributes for parser declarations (which were part of the "standard" module) were being supplied by the Copper grammar, which is a violation of the rules. There are three modules here: the "standard" syntax module with the production, the "translation" module with the attribute occurrence, and the "Copper" module, where this equation existed. This declaration does have some "generic" parts: it indicates that a parser should be built, and it declares a new function in the environment. However, part of the semantics of this declaration is simply Copper-specific: somehow the function implementation calls the Copper-generated parser. Our solution was to move this parser declaration entirely out of the "standard" module and into the "Copper" module.

Some parts of the AG language and this analysis were motivated by our attempts to get Silver to conform to the restrictions. We found that we were abusing forwarding, using it as a way to define default values for attributes where the forwarded-to tree was not, in fact, semantically equivalent to the forwarding tree in the slightest. This was our original motivation for introducing default equations, to remove these abuses of forwarding.

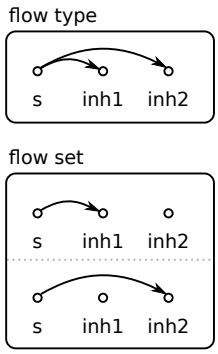
There were two sorts of negative changes made to the Silver specification in order to make it pass the analysis. The first of these resulted from the conservative rules for handling reference attributes. On two occasions, inherited attribute equations had to be supplied whose values are never actually used. In one case, a nonterminal that represents information about concrete syntax has two synthesized attributes, one for

a normalization process and one for translation. These attributes have different (true) inherited dependencies, but because they internally both made use of references, the analysis required the full set of inherited attributes be supplied in order to access either synthesized attribute.

The second sort of negative changes involved introducing workarounds for code that we already knew needed refactoring, but we did not want to fix, yet. In fact, in many of these cases, the analysis lead us to code that already had "TODO" comments complaining about a design for reasons unrelated to the analysis. The most significant of these is the use of a single nonterminal (called `DclInfo`) as a data structure to represent several different types of declarations in Silver (attributes, types, values, occurs-on, etc.) This is a legacy from when Silver did not have parametric polymorphism and our environment needed to pick a single monomorphic type to return as a result of any query. We could fix this issue by parameterizing our environment lookup functions, and introducing separate types for each namespace (`TypeDclInfo`, `AttrDclInfo`, etc.) But to forgo fixing this issue for the time being, we introduced "error" equations for attributes that did not have sensible values otherwise (e.g. attributes that make sense for *value* declarations but do not apply to *type* declarations.) These directly silence the analysis and leave a potential problem (if these equations were to be demanded), but they're visibly noisy about this problem in the code and the use of "error" equations remains statically detectable. These error equations are essentially a form of "technical debt" - legitimate problems that we will change later, but for various reasons decide not to do just yet.

In the end, most of the changes necessary were to the host language itself, and the extensions then passed with minimal further effort. The translation code was identified as a pure extension to the host language, which we found surprising initially.

```
production swap
l::A ::= r::A
{
 r.inh1 = l.inh2;
 r.inh2 = l.inh1;
 l.s = r.s;
}
production left
l::A ::=
{
 l.s = l.inh1;
}
```




```
aspect swap
l::A ::= r::A
{
 l.op = r.op;
}
aspect left
l::A ::=
{
 l.op = l.inh2;
}
```

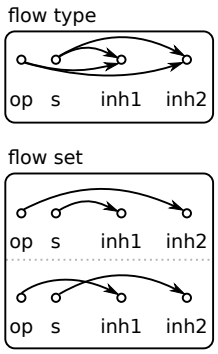


Figure 5.17: An example of the difference between flow types and flow sets. On the right, the consequences of an extension.

Our guess, prior to developing this analysis, was that a translation would somehow be involved enough to need to be considered part of the host language. This was not the case. In retrospect, we have preserved the ability to write just about any analysis over the host language that we want, translation just being one particular kind.

5.8 Extending to circularity

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

In figure 5.17, we show a small example grammar (and extension) demonstrating the differences between flow types and flow sets. For a flow type, we care only about the potential dependencies, while for flow sets we care about how multiple attributes

relate to each other, in order to more accurately understand how information flows. Notice how, on the left, we see that the attribute `s` never (in fact) depends on both inherited attributes at once. Also take note, on the right, that the flow sets we get are correlated: we know that `s` and `op` never depend on the same inherited attribute.

Now consider, as a hypothetical example using the extension on the right, if we wrote something like `r.inh1 = r.op` and then accessed `r.s`. This would be considered a circularity using the flow types, because `r.op` potentially depends upon `r.inh1` and `r.s` potentially depends on `inh1`. However, using the flow sets instead, we can see two cases: either `r.s` depends on `inh2` (and we don't access `inh1`) or `r.s` does depend on `inh1` but *in that case* `r.op` does not depend on `inh1` and so there is no circularity. This precision is why flow sets are desired for the circularity analysis.

For the completeness portion of the analysis (everything described previously in this chapter,) we still use the flow type, and so we do not see any differences there. However, we will now obtain flow types as the join of the flow set for each nonterminal, since we must now compute those. Returning to the algorithm for computing flow types, in order to compute flow sets instead, we have three major complications:

1. Each nonterminal stitch point will have multiple possible flow graphs to stitch in (each element of the corresponding flow set) instead of just one, and all possible combinations for all stitch points in the production must be used, yielding multiple production flow graphs per production.

2. Each projection stitch point will have multiple possible production flow graphs that can be used as a result. All production flow graphs from the last iteration are used as the possibilities for the current iteration. For the first iteration, just the static production graph is used. (Note that if a production contains a

match against itself, this presents no added difficultly for the flow computation. We simply use last iteration's set of production flow graphs, just as we would previously use the last iteration's single production graph.)

3. We now need to know which flow sets are induced by the host language alone, and which come as a result of introducing an extension. As a result, we can not longer use to "suspect edges" trick to perform the flow computation just once (and handle circular dependencies between modules.) Instead, we need to compute it for the host language, and then again for the extended language. (We consider the collection of all imported modules "the host language" and module being analyzed as "the extension.")

We accomplish "modularizing" the non-circularity test by the same basic method as for flow types. We fix the permissible elements of the flow sets in the host language, and check to ensure that extensions introduce no flows that are not already permissible. And finally, we perform the usual non-circularity test, checking all generated production flow graphs to ensure there are no cycles.

The tricky part of this is determining what "no flows that are not already permissible" means. With flow types, this is trivial, as each synthesized attribute is in its own little world, independent of other attributes, and can be considered "extension" and "host" in isolation. As a result, we can just look at each synthesized equation and ensure its inherited dependencies are a subset of the corresponding flow type, while allowing that flow type value to be determined by whatever module introduces that synthesized attribute. For flow sets, however, we're capturing non-local relationships between attributes, and this is no longer so obvious.

But by computing host and extension separately, we're able to solve this problem

rather handily. Each flow graph consists only of edges from synthesized attributes to inherited attributes. We simply take each extension flow graph, truncate out the extension synthesized attributes (and edges from them,) and then check to ensure the resulting flow graph is a subset of an existing host flow graph. Thus, each flow induced by an extension is the same (or simpler) than an existing flow that can happen in the host language (modulo extension synthesized attributes, where the extension is authoritative.) Returning to figure 5.17, we can see that when `op` is removed, each flow graphs on the right (in this case) is exactly the flow graph on the left. This makes sense, as all this extension really did was add another synthesized attribute, and so we shouldn't see any changes in flow behavior.

5.8.1 Discussion

We do not consider non-circularity to be a part of our modular well-definedness analysis, for a number of different reasons.

The most important reason is that the flow sets the circularity analysis associates with nonterminals are not a clean interface. They are a very complex one. We can reasonably infer from a host language most flow types, with a few exceptions where the host would prefer extensions be allowed to depend on an attribute but the host language never does. The flow sets, however, are vastly more difficult to reason about (as they arise non-locally), much more difficult to specify, and result in error messages that would best be described as unhelpful. We no longer have a simple answer to what inherited attributes a synthesized attribute equation is permitted to depend upon. There may be a set of very subtly different flows which an extension developer may run afoul of, despite being well within the flow type. Further, these flows are global properties of an attribute grammar, and so our error messages cannot

necessarily pinpoint which piece of code caused the problem to appear, only that a particular production now has a problem. This makes understanding the restrictions (and understanding what went wrong in violating these restrictions) very difficult.

Another reason to avoid non-circularity is that it is much more difficult to compute. The flow set analysis potentially blows up exponentially, while the flow type analysis is much simpler. If we believed the non-circularity analysis were an important part of ensuring that extensions would work, this might be worthwhile. However, there are a number of reasons to believe it is not important. For one, we have never actually encountered a circularity problem with an extension that wasn't also a flow type problem. Flow types by themselves are quite good at preventing circularity issues, because they permit only certain circularities to exist. If a `translation` attribute can only depend upon `env`, then we can only introduce a circularity there by using `translation` of a subtree to define `env` for that subtree. This restriction is sufficient to make potential circularities rather obvious when they are written, as well as prevent accidental circularities (that are usually a result of accidentally emitting very wide dependencies, which would likely violate the flow type.)

Finally, there are a few other reasons not to bother. For one, because Silver is a lazy language, circularities can actually be productive, if they compute streams for example, or other data structures where only partial demands may be made. (Silver comes with a pretty printing library that uses a circular stream of this sort.) Or the circularities may be entirely false anyway (especially because our method of dealing with references is extremely conservative). Finally, circularity is just one kind of nontermination, and to close this gap we must also check for termination of higher-order attribute expansion [65] (via locals and forward equations) and well as general function termination. Some of these problems may be solvable, but we

consider it reasonable to call them beyond the scope of this thesis. As a result, for our purposes in developing reliably composable language extensions, we confine the modular well-definedness analysis to merely refer to modular effective completeness, which we consider sufficient.

5.9 Related work

Knuth provided (and later corrected) a circularity analysis when introducing attribute grammars [20]. In presenting higher-order attributes, Vogt et al. [56] extended Knuth's completeness and circularity analyses to that setting. Reference and remote attributes do not have a precise circularity analysis [59], as the problem is undecidable. Completeness in these settings is simply a matter of using occur-on relationships to check for the existence of all equations for all attributes. With forwarding, flow-analysis is used to check completeness and thus a *definedness analysis* that combines the check of completeness and circularity was defined [21, 72]. This analysis used *dependency functions* instead of flow graphs in order to distinguish between synthesized attributes that depend on no inherited attributes and those that cannot be computed because of a missing equation or circularity, and thus conflate these two types of errors. All of these are non-modular analyses.

Saraiva and Swierstra [79] present *generic attribute grammars* in which modules can be parameterized by nonterminals with a particular flow type. This is the origin of flow types, though we use them for slightly different purposes, and we infer them rather than specify them as part of the module. Generic AGs are very different from our language extension model, however. It does not allow for multiple independent extensions to be composed, except by first merging them into a single extension, on

which the analysis must then be performed, effectively making it monolithic.

In AspectAG, Viera et al. [10] have shown the completeness analysis can be encoded in the type system of Haskell. However, this analysis is again performed at the time of composition (by the type checker) and is thus a monolithic analysis.

Current AG systems such as JastAdd [81] and Kiama [9] do not do static flow analysis but, like previous versions of Silver, instead provide error messages at attribute evaluation time that indicates the missing equation or circularity. An extension writer can write test cases to test his or her specification and perhaps find any lurking problems, but this does not provide any assurances if independently developed grammars are later composed. And of course, no one has written tests for an attribute grammar composed of several independent extensions.

Chapter 6

Non-interference

In the previous chapter, we have described an analysis that ensures composition of language extensions will always result in a well-defined attribute grammar. This ensures we are always able to compose extensions together, without errors occurring during the composition process. However, we are left knowing only fairly weak properties about the behavior of the resulting composed language. The trouble is that attribute values can now be computed by one independent extension and consumed by another, and it is quite easy to imagine ways in which this can result in undesirable behavior, despite the lack of type errors or well-definedness issues in the attribute grammar.

We will take two different perspectives on this problem throughout this chapter. The first perspective views the problem as undesirable interaction between language extensions, which we call *interference*. From this perspective, we are concerned with the extension developer's task in ensuring their extension will continue to work when composed with other unknown extensions. Although this perspective makes clear the problem we wish to avoid, it does not leave us with much guidance on how to resolve the problem.

The second perspective is to consider modular and composable proofs of properties about attribute grammars. One major difference with this perspective is that now the host language, too, is involved. We wish to prove our host language and extensions

correct, and be sure these properties will hold of the composed language, too. In this way, we can ensure non-interference: if the proofs of correctness still hold, then the extensions behavior should be as expected. The most important development we make with this perspective is the notion of *coherence*, which is the particular tool we use to ensure our properties and proofs will not be invalidated when other extensions are composed into the system.

We are far from the day when our compilers are routinely verified, however, and so instead of doing verification, we wish to use this perspective as a theoretical framework for a more practical way of achieving non-interference. Throughout this chapter, we will weave these two perspectives together to develop a practical, testing-based approach to ensuring extensions are non-interfering, grounded in the notion of coherence. This allows us to identify potential interference problems in a quick and practical way, without the need to actually do verification.

We begin in section 6.1 with a more detailed explanation of the interference problem. We show two examples of language extensions that seem correct in isolation, but when composed together show observable errors. However, this perspective seems to give us no guidance on what went wrong: the problem just looks like something that needs glue code to resolve, what could the extension developers have done?

In order to find a solution, we shift attention to the verification perspective. In section 6.2, we take a look at what proofs of properties about attribute grammars look like. In particular, we consider how modular proofs can be constructed. Then in section 6.3, we introduce a *coherence* meta-property (that is, a property we can show about the properties we prove about attribute grammars). Under the assumption that language extensions do not violate coherent properties, we show how proofs of coherent properties can be automatically extended to cover the new cases arising from

composition with other language extensions. As a result, coherent properties can be proved for individual language extensions (and the host language) in isolation and remain true for their composition with other extensions.

We then justify that assumption, in section 6.4, by showing a set of restrictions that are sufficient to ensure an extension preserves all coherent properties, thus ensuring non-interference. While we consider these restrictions a reasonable burden on extension developers (enforcement can even be done syntactically, without the need to actually do verification), they are unreasonably restricting to the capabilities of extensions. And so in section 6.5 we develop an "attribute properties" approach to ensuring coherent properties are preserved. This is sufficient to solve the problems we identified in the overly restrictive approach.

Finally, in section 6.6, we describe a method for enforcement of this "attribute properties" approach using randomized property testing—managing to entirely avoid having to do verification in practice. In section 6.7, we apply this technique to a real compiler, providing some evidence that the testing method works well enough. We conclude with some related work in section 6.8 and some discussion in section 6.9. In particular, we note that although the testing approach is less perfect than verification, there are several reasons why it might work better than one might initially expect. Lastly, we note that our theoretical development has left us with a notion of blame: even if an interference bug slips through testing, we are able to identify the extension at fault, and problems in the composed language are not emergent behavior with no solution.

6.1 The problem

The problem of interference between composable extensions arises because the developers of independent language extensions (E_A and E_B) are unable to examine the composed language $H \triangleleft (E_A \uplus_\emptyset E_B)$. Each artifact for H, $H \triangleleft E_A$, and $H \triangleleft E_B$ are whole programs about which their developers can reason or write tests. It may not be possible to construct any trees in either individual extended language (that is, $H \triangleleft E_A$ or $H \triangleleft E_B$) that demonstrate any flaws, but we may be able to do so for the composed language. Indeed, it may be difficult to precisely identify what "flaw" means, as no one has necessarily developed a semantics for the composed language specifying how the two extensions should interact. Worse still, having found a tree that reveals (what we have decided is) a flaw in the composed compiler, there may not be any obvious way to fix it. Both extensions may seem perfectly innocent in isolation, and the flaw may be the result of an unfortunate interaction that seems the fault of neither or seems to require glue code to fix. When extensions are developed independently, the differing developers may be quite willing to simply point their fingers at each other, resolving nothing.

For example, consider figure 6.1, showing two extensions to a Boolean expression language (like that from figure 3.5 for reference, but referring back is likely unnecessary). Each extension in this figure introduces some syntax and some associated synthesized attribute for analysis (aspects for `or` and `literal` shown, the rest omitted). E_A attempts to discover the use of unsanitized input (e.g., in a normal programming language, to detect SQL-injection vulnerabilities) by introducing a `taint` annotation on expressions, as well as an analysis for discovering whether tainted values are used

```
production taint
e::Expr ::= x::Expr
{
  e.is_tainted = true;
  forwards to x;
}
aspect or
e::Expr ::= l::Expr  r::Expr
{
  e.is_tainted =
    l.is_tainted || r.is_tainted;
}
aspect literal
e::Expr ::= b::Boolean
{
  e.is_tainted = false;
}
```

```
production identity
e::Expr ::= x::Expr
{
  forwards to x.id_transform;
}
aspect or
e::Expr ::= l::Expr  r::Expr
{
  e.id_transform =
   or(l.id_transform, r.id_transform);
}
aspect literal
e::Expr ::= b::Boolean
{
  e.id_transform = literal(b);
}
```

Figure 6.1: A simple example of interference. Left: E_A. Right: E_B.

in a subexpression. That is, we expect

`or(literal(false), taint(`t`)).is_tainted`

to discover the tainted subtree, for any t. Extension E_B does something seemingly useless, but also perfectly innocent: it transforms an input tree (in the host language) to itself. It does this by way of a synthesized attribute on expressions that recursively reconstructs the same expression. (Although `identity` seems useless, it is the simplest of tree transformations, which are generally quite useful.)

Because this E_B transformation is only defined on host language productions (and is unaware of other extensions like E_A and so cannot handle `taint` except via forwarding) this attribute has the effect of replacing forwarding productions with what they forward to. In the figure, we see that the implementors of E_A have their `taint` production simply forward to whatever expression it wraps. Thus, the analysis's success depends on their `is_tainted` analysis being applied to the forwarding production.

However, this means there is trouble for an expression like

```
identity(or(literal(false), taint(t))).is_tainted
```

which is the same tree as the last example, except with `identity` in between the tree and asking for the `is_tainted` result. Here, `identity` (which should do nothing) will essentially transform away the reference to `taint`, leaving the `is_tainted` analysis nothing to discover, even though it should be discovering a tainted subexpression in this case.

Each extension works perfectly in isolation: any number of tests we might write for them will all discover no observable problems. But the trouble here is not *just* that there is a problem with the composed language that is undetectable in isolation. It's also that we have no guidance on how this problem should be resolved. Did E_A err in forwarding to its wrapped expression for the taint annotation syntax? Did E_B err in transforming away the other extension's syntax? Both? Something else? Without some sort of solution to this kind of interference problem, language extension could be a siren song: a tool seemingly useful in toy experiments, but perhaps fraught with more problems than it's worth when used in the real world. After all, the space of potential interference grows exponentially with each new extension in the ecosystem. Interference can easily start off seeming like a non-issue, and then suddenly seem like an insurmountable problem.

6.2 Reasoning about attribute grammars

Since we cannot see an immediate solution to the problem of conflicting language extensions above, we need to shift perspectives. Indeed, part of the problem is that we don't even know exactly what went wrong. We claim each extension works perfectly

in isolation, but what does that mean exactly? And while it seems like the composed language is misbehaving, how precisely should we define good behavior?

So let us instead think in terms of properties of the language and attribute grammars. If we prove the host language, or an extension, works correctly according to the specification of its semantics, then we want to be confident that composed language will also satisfy those same properties.

6.2.1 Properties of languages and modular non-interference

Our model for verified language extension is for each module (the host language and each extension) to come with an associated set of correctness properties for that module. Each module will also prove these properties hold for their attribute grammar. For example, with the `is_tainted` extension we showed previously, we might wish to show:

$$\forall t.\ \texttt{taint(}t\texttt{).is\_tainted} = true$$

as a very simplistic property (really more of a test case, but even a property this simple will prove illustrative later.) More generally, the host language H will have a set of properties $\mathscr{P}(H) = \{P_1^H, P_2^H, ...\}$, and proofs that each of these properties holds, i.e. proofs of $\forall t \in H.\ P_i^H(t)$ for each i. Similarly, an extension E_A will have a set of properties $\mathscr{P}(E_A) = \{P_1^A, P_2^A, ...\}$, and proofs of the form $\forall t \in H \triangleleft E_A.\ P_i^A(t)$. Note that we have expanded the range of quantification here to the extended language, not just the host. An extension will also contain all properties the host language contains, so $\mathscr{P}(E_A) \supseteq \mathscr{P}(H)$. This means the extension must contain proofs of $\forall t \in H \triangleleft E_A.\ P_i^H(t)$ (i.e. proofs of host language properties but quantified over extended language trees.)

Our approach to eliminating interference is for all of these properties to remain true of a composed language. That is, given n language extensions, we want the following to be true:

$$\forall i \in [1 \ldots n], P^i \in \mathscr{P}(E_i), t \in H \triangleleft \{E_1 \uplus_\emptyset \cdots \uplus_\emptyset E_n\}.\ \ P^i(t)$$

Put another way, every property from each smaller extended language should hold of the composed language that includes all those extensions. Obviously, this requires some sort of restrictions to accomplish.

As is our running theme, we wish to achieve this in a modular way. We name the modular restriction that accomplishes this goal *noninterfering*, and we want this restriction to be such that the following holds:

Theorem 6.2.1 (Modular non-interference)**.**

$$\begin{aligned}\big(\forall i \in [1,n].\ \mathit{noninterfering}(H \triangleleft E_i)\big) \implies \\ \forall i \in [1,n], P^i_j \in \mathscr{P}(E_i).\ (\forall t \in H \triangleleft E_i.\ P^i_j(t)) \implies \\ \forall t \in H \triangleleft \{E_1 \uplus_\emptyset \cdots \uplus_\emptyset E_n\}.\ P^i_j(t)\end{aligned}$$

We will prove this theorem later, after we have a concrete definition for *noninterfering*. The main feature of the above goal is that we have ensured each extension E_i satisfies *noninterfering* in isolation. After that, we are able to turn these proofs of properties in isolation into proofs of the same properties, now quantified over the whole composed language. In other words, knowing only that each extension is "non-interfering," we must know that all of its correctness proofs will still hold of the fully extended language. When the correctness properties for each extension are proven to hold for the composed compiler, we have eliminated the possibility of interference sneaking into

the composed compiler. Indeed, we would have proof that each extension is behaving exactly as specified.

One piece of this to take note of is how we apply *noninterfering* to each extension attribute grammar $H \triangleleft E_i$. Unlike the previous chapter, where we took special note of module relationships, we actually do not care where anything comes from for this analysis. Indeed, it is entirely possible for the host language itself to fail this analysis (and thus no extensions could be safely written for it.) And so host language developers, too, should take care to ensure *noninterfering*(H).

As a final note, this definition of non-interference gives us a precise notion of blame, despite not even knowing yet how to define *noninterfering*. If an extension is non-interfering, then correctness proofs for different extensions can be automatically extended over it. If a bug exists in the final, composed artifact, then either one extension is interfering (that is, *noninterfering* does not hold of it), or one extension's specification was incorrect. Both of these possibilities can be observed in isolation from other extensions. Thus, although the end-user may not be able to diagnose a problem, the extension developers involved will not be conflicted about who is to blame for a bug.

6.2.2 Induction on decorated trees

There are a couple of important points of formality that are best noted here, though we do not come to use them until later. The above definition of modular non-interference glosses over a few subtleties.

First, what are properties? For now, we'll take a relatively simplified (or constrained) view.

Definition 6.2.2 (Properties, for our purposes)**.** A property over a tree is a logical proposition with a free (tree-typed) variable. The proposition may be composed of:

1. simple logical connectives (and/or/implies),
2. quantification over non-tree types (such as numbers),
3. the use of inductively-defined relations over trees (such as equality), and
4. the use of attribute evaluation on trees (which we call *evaluation relations*).

Note that evaluation relations *are* inductive relations, but these come from the equations written in the attribute grammar, instead of being defined as part of the verification, and so we distinguish them. We'll discuss relations a bit more shortly (and see example properties.) Note that for now, only the free variable ranges over trees. Later on in section 6.3.3, we generalize slightly to n-ary relations and propositions of multiple tree-typed variables, and connect these generalizations back to just talking about properties.

Second, we talk about properties P that range over one language, and then conflate them to also apply to an extended language. For instance, the claim that $\mathscr{P}(E_A) \supseteq \mathscr{P}(H)$ seems like a type error, since the properties range over different languages ($H \triangleleft E_A$ and H alone respectively). We will fix this issue soon in section 6.3, but we wanted to point out this imprecision up front.

Third, we need to be careful about the nature of quantification for these properties. We'll often write the properties in closed form (i.e. as propositions in the form $\forall t \in H.\ P(t)$), in order to show a range of quantification. But although this looks as if we're quantifying over a language, no property actually does that (nor do we have a definition of what that means). In truth, H is just a stand-in for a choice of

nonterminal from the language H that the variable actually ranges over, so that we can talk about properties for a language generally.

We overload this notion of quantification over "languages" a little bit more as well. One of the things we need to do (later) is take properties stated as $\forall t \in H \triangleleft E.\ P(t)$ and instead apply them over the language H, but this is nonsensical if the property refers to attributes from E. Note that our overall concern here is composing a host language (H) and several extensions (E_i) into a composed language (L). The least problematic way to interpret $\forall t \in H$ and $\forall t \in H \triangleleft E$ is to only ever quantify over the composed attribute grammar (L). We should then interpret the *stated* range of quantification (H or $H \triangleleft E$) as being a restriction to only productions from the specified modules. That is $\forall t \in H.\ P(t)$ would more formally be something like $\forall t \in L.\ host(t) \implies P(t)$, where *host* indicates that only productions from H are present in t (and of course L is still a stand-in for a particular nonterminal). It is possible to change the range of quantification (from H to L) freely like this because extensions are not able to *modify* existing equations in the attribute grammar. A proof of a property about H will still be true of L restricted to productions from H because it's the same productions with the same arguments about the same equations computing the same values. We only potentially invalidate a proof when introducing a new (necessarily forwarding) production to an existing nonterminal, which introduces a new case to worry about.

Finally, and perhaps more importantly, the way we prove properties about attribute grammars is by induction on *decorated* trees. We have previously described decorated trees as undecorated (i.e. syntax) trees that have been supplied with a set of inherited attributes, on which we are thus able to compute synthesized attributes. This is a perfectly fine operational view, but for the purposes of reasoning we should

```
production
e::Expr ::= l::Expr  r::Expr
{
  local loc :: Expr = ...
  forwards to ...
}
```

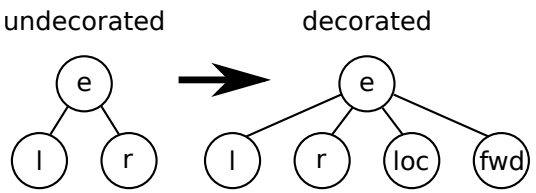


Figure 6.2: From undecorated to decorated trees, on which we reason.

expand upon it a bit. For reasoning, we will look at a decorated tree as a tree where *every decoration site* within a production becomes a child of each corresponding decorated node. This includes not just every ordinary child of the original undecorated tree (now also decorated nodes,) but all other places where undecorated trees are supplied with inherited attributes, including each local and the forward tree, if present. This is illustrated in figure 6.2. Just like with undecorated trees, each production gives rise to a different kind of node in the tree.

The benefit of induction on decorated trees is that, when trying to show a property holds for the case of one production node, we assume the induction hypothesis for all of its child decoration sites, which includes the forwarded-to tree (when the production is a forwarding one) and not just its undecorated children. This ability to assume the induction hypothesis for the forward tree will prove a critical part of our approach. For this form of reasoning to be acceptable, of course, it must be the case that these decorated trees are finite (and no true circularities exist in attribute dependencies, or in other words, that the attribute grammar always terminates). In principle, this may not be the case, as forwarding and local equations can lead to essentially infinite expansion. But for our purposes, we will simply assume finiteness and termination.

For a typical attribute grammar and for typical properties, we will state propo-

```
nonterminal RepMin with min;
syn min :: Integer;
production inner
e::RepMin ::= x::RepMin
              y::RepMin
{ e.min = min(x.min, y.min);
}
production leaf
e::RepMin ::= x::Integer
{ e.min = x;
}
```

Interesting relations:

$\texttt{lte\_all} : \mathit{RepMin} \to \mathit{Integer} \to \mathit{Prop}$

$\texttt{lte\_all}(\mathit{inner}(x, y), m)$ `:-`
$\quad \texttt{lte\_all}(x, m) \wedge \texttt{lte\_all}(y, m)$

$\texttt{lte\_all}(\mathit{leaf}(x), m)$ `:-` $m <= x$

$\texttt{exists} : \mathit{RepMin} \to \mathit{Integer} \to \mathit{Prop}$

$\texttt{exists}(\mathit{inner}(x, y), m)$ `:-` $\texttt{exists}(x, m) \vee \texttt{exists}(y, m)$

$\texttt{exists}(\mathit{leaf}(x), m)$ `:-` $x = m$

`min_correctness` property: $\forall t, m.$
$t.\texttt{min} = m \to \texttt{lte\_all}(t, m) \wedge \texttt{exists}(t, m)$
Proof: by induction on the decorated tree t.

Figure 6.3: Example of inductive relations and properties we might try to show of an attribute grammar.

sitions about each nonterminal, and prove them by mutual induction on all relevant nonterminals. One thing to note is that the properties we're usually trying to prove nearly always involve new inductively-defined relations on the tree. That is, relations defined by case analysis on the production at the root of a tree. Examples of this appear in figure 6.3. (This example is based on the RepMin example of figure 3.3, but we reproduce this grammar on the left of the figure, so referring back is unnecessary.) We define new relations `lte_all` and `exists` which exact different requirements on `leaf` and `inner` nodes. (We use the reverse implies notation (`:-`) to better show how each production maps to its consequents.)

The key characteristic of these sorts of relations is that they map each production to a property that should hold for corresponding nodes in a tree. The two relations shown are the primary components of the correctness property `min_correctness` stated below them in the figure. We will be paying special attention to these kinds of relations. This property also makes use of the evaluation relation for `min`, which is necessary to make any claims about the values of attributes, but we write this

with attribute grammar notation ($t.\texttt{min} = m$). In the figure, we omit the proof of `min_correctness`, as it is both relatively simple (just two cases: one for each production), and it is not our intention to dive too deeply into the details of proving properties about attribute grammars. However, we do show this example formally (using Coq) in appendix A, for the curious.

6.2.3 Extending proofs to extended languages

We wish to briefly consider what is involved in extending a proof about a host language to an extended language. In other words, if we had a fully verified host language, and we introduced a language extension, what is involved in modifying the verification to accommodate the extension? Note that this does not yet consider the much harder problem of composing extensions together, we just want to look at what is involved in extending a verification alongside extending an attribute grammar in the context of a single extension. In figure 6.4, we describe a simple extension to our RepMin example. (Once more, we show this example more formally in appendix A.)

Our "host language" in figure 6.3 only dealt with introducing the two types of internal nodes (which we called `inner` and `leaf`), and the `min` attribute computing the minimum of the tree. We can now extend this to include two new attributes, as well as a novel production (not typically present in RepMin): `three`, which simply has three children. The first attribute `global_min` is an inherited attribute, intended to pass down the global minimum value of the tree for use in the second new attribute `rep` which replicates the tree structure with all values replaced by this minimum. Assigning `global_min` the value of `min` is typically accomplished from a root nonterminal (as in figure 3.3,) but this is not a core part of our concern here, and so we omit it. We have introduced a new production and new attributes, and we also

```
production three
l::RepMin ::=
  x::RepMin  y::RepMin  z::RepMin
{
  l.min = min(min(x.min, y.min),
              z.min);
  forwards to
    inner(inner(x, y), z);
}
syn rep :: RepMin;
inh global_min :: Integer;
aspect inner
e::RepMin ::= x::RepMin
              y::RepMin
{ e.rep = inner(x.rep, y.rep);
  x.global_min = e.global_min;
  y.global_min = e.global_min;
}
aspect leaf
e::RepMin ::= x::Integer
{ e.rep = leaf(e.global_min);
}
```

Extensions to existing relations:

$\texttt{lte\_all}(three(x,y,z),m)$:- $\texttt{lte\_all}(x,m) \wedge$
$\quad \texttt{lte\_all}(y,m) \wedge \texttt{lte\_all}(z,m)$
$\texttt{exists}(three(x,y,z),v)$:- $\texttt{exists}(x,v) \vee$
$\quad \texttt{exists}(y,v) \vee \texttt{exists}(z,v)$

New relations:

$\texttt{eq\_all}(three(x,y,z),m)$:- $\texttt{eq\_all}(x,m) \wedge$
$\quad \texttt{eq\_all}(y,m) \wedge \texttt{eq\_all}(z,m)$
$\texttt{eq\_all}(inner(x,y),m)$:-
$\quad \texttt{eq\_all}(x,m) \wedge \texttt{eq\_all}(y,m)$
$\texttt{eq\_all}(leaf(x),m)$:- $x = m$

New property:

`rep_all_equal`:
$\forall t, m, r.\quad t.\texttt{global\_min} = m \rightarrow$
$\quad t.\texttt{rep} = r \rightarrow \texttt{eq\_all}(r,m)$

Proof: by induction on the decorated tree t.

Figure 6.4: An example of an extension, showing extension productions, equations, attributes, relations, proofs.

introduce a new relation `eq_all` and a new property `rep_all_min` showing (part of) the correctness of `rep`.[1]

When extending a language, the verification may also need extending in a number of ways:

- The host language specification itself needs extending. In the upper right of the figure, we show new cases of relations that were originally defined as part of the verification of the host language, which now have to also handle the production

[1] Typically, there is also a requirement for structural equality between the original tree and `rep`, but this doesn't demonstrate anything else interesting for our purposes here, so we also omit it.

`three`.

- Any proofs of properties from the host language will have new cases to show for new productions. This is not shown in the figure (though visible in appendix A), but the proof from figure 6.3 will require a new case for `three`.

- Extensions may choose to introduce new properties and relations defined inductively on nonterminals of the host language, along with new proofs showing new properties about trees. (Such as the definition of `eq_all` and proof of the associated property `rep_all_equal`.) These can be properties about the unextended language that the extension wishes to make use of, but the host language hadn't shown itself. (No such property is present in this example, but in principle the `min_correctness` property may not have been in the host language, and could be present here instead.)

- For new nonterminals introduced by the extension (also not a part of this example), there may also be new properties and new proofs.

In isolation, the extension developers are perfectly capable of manually extending the definitions and proofs and introducing the necessary new ones, as this example demonstrates. Extension is not the problem. All of our troubles stem from the need to do automatic composition of independent extensions. As soon as we have two independent extensions, each developer no longer knows about the other's new productions, nor do they know what new properties other extensions might be relying on. Worse still, there's seemingly a creative component: specifications themselves have gaps, not just proofs. When a property is introduced independently from a new

production, we don't have a complete specification. Somehow, we need to be able to complete these automatically.

6.3 Coherence

We solve the problem of completing both specifications and proofs by introducing the concept of *coherence*. This allows us to write modular proofs about extensions, and be confident they hold for a language composed with other extensions. To start with, coherence gives us a mechanism to automatically complete the gaps in specifications that arise as a result of composition of multiple language extensions. This comes at the cost, however, of limiting what properties extensions can state about the attribute grammar. This also limits what kinds of extensions are possible, if we require coherence. If the specification of an extension's correct behavior is inherently incoherent, then that extension is no longer permissible.

Coherence can be thought of as requiring the "semantic equivalence of forwarding." Properties that are not equally true of a forwarding tree and what it forwards to are considered to be incoherent.

Definition 6.3.1 (Coherent properties)**.** A property P over a language L is coherent if

$$\forall t \in L.\ P(t) \iff P(t.forward)$$

Where $t.forward$ is the tree that the root production of t forwards to (or if the root is non-forwarding, then $t.forward = t$.) In other words, coherence requires consistency between the properties that hold of a tree rooted in a forwarding production and the tree it forwards to.

To make it easier to refer to each direction of this notion of coherence, we give them names by analogy to logic. We call the $P(t) \implies P(t.forward)$ direction *soundness*, and we call the $P(t.forward) \implies P(t)$ direction *completeness*. In essence, we view the host language as the authority on semantics, and the extension as merely a different formal syntax, which must get meaning solely through the semantics of the host language via the forwarding equation. In this way, we are finally being precise about what it means to forward to a "semantically equivalent tree"—a phrase used even in the original paper on forwarding, but has been without a precise definition until now.

Coherence gives us an automatic and natural means of dealing with the gaps in specifications that arise as a result of composition. These gaps are created because relations defined over nonterminals require new cases to handle any new forwarding productions from other extensions. Because any coherent property should be equally true of a tree and what it forwards to, this gives us an automatic method of determining what should hold of a new forwarding production for an existing relation. Unary relations are just one kind of property, and so a unary relation is coherent according to the above definition of a coherent property. So let us begin with unary relations:

Definition 6.3.2 (Coherent extension of unary relations)**.** Given a coherent unary relation R over a language H, we can automatically extend this relation to range over an extended language $H \triangleleft E$. We leave the definition of $R(t)$ alone for all cases where t is a non-forwarding production, and for each forwarding production define it as being equivalent to $R(t.forward)$. Thanks to our finiteness and termination assumptions, this is well-defined because recursive expansion of $R(t.forward)$ will ultimately terminate in non-forwarding productions from the host language, a restricted subset of

cases where we already have a complete definition of R. The resulting relation is still coherent, if all of its dependent relations are still coherent.

As an example[2], one of the missing cases for a relation that we manually filled in back in figure 6.4 was for the `three` production, which forwarded to `inner`, and so we'd have the following coherent extension:

$$\texttt{lte\_all}(\mathit{three}(x, y, z), m) \texttt{ :- } \texttt{lte\_all}(\texttt{inner}(\texttt{inner}(x, y), z), m)$$

This defers what `lte_all` means on `three` to the tree that `three` forwards to, instead of the extension having to manually specify it. For this to be well-defined, all we need to know is that expansion of the decorated tree eventually terminates (which means we will eventually arrive at host language productions, even if there are more layers of forwarding productions in between). The complexity and behavior of the forwarding equation is irrelevant: it can analyze or ignore its children, use higher-order attributes, or otherwise consist of any valid expression. In this example, expansion stops immediately with two `inner` non-forwarding productions, which is an especially simple case.

If we are able to coherently extend all the relations in a property, then (we will soon prove) this is enough to coherently extend the property itself. As a result, we are able to freely take any coherent property P over a language H, and sensibly speak of that "same" property (really its coherent extension) holding over an extended language $H \triangleleft E$. In our definition of modular non-interference (previously in section 6.2.1), we already had need of this capability when making claims about the properties that extension developers needed to show about their extended language. We required

[2]Well, we're immediately using a non-unary relation with this example, but the second parameter is a primitive type, so it still fits with our limited notion of property.

that extension developers show that the host language's properties still held of their extended language. More precisely then, we are actually requiring that extension developers show the coherent extension of host language properties.

Indeed, extension developers could choose to explicitly define new relations *only* concerning non-forwarding productions, allowing the forwarding cases to filled in by coherent extension. This would actually guarantee that the relation is coherent (assuming it does not rely on any other incoherent relations). However, this is also the least useful way to do verification of a language extension. If we do not write specifications for forwarding productions, then the forwarding equation itself (which is just code, from the program we're trying to verify) becomes the specification. Usually with verification, we try to make claims more involved than "the code does what the code says it does." The only route to checking whether our forwarding equation is correct is to make claims about the meaning of a forwarding production. We must then show those claims are coherent.

6.3.1 Examples of incoherence

In light of this new notion of coherence, let us revisit the examples of interference from section 6.1. For `is_tainted` (from figure 6.1) we have a problem with the `taint` production that forwards to its child and the synthesized attribute that does the taint analysis. Recall the extremely simple example property we might wish to show:

$$\texttt{taint}(t).\texttt{is\_tainted} = \texttt{true}$$

This property turns out to be incoherent for two reasons.

Unfortunately, the first reason is related to a technical problem with its implicit use of equality on trees (a topic we will get to later, in subsection 6.3.4). Let us

rephrase this somewhat and instantiate the quantification with an example, in an attempt to side-step this issue for now, and observe the more interesting problem:

$$t.\texttt{is\_tainted} = \texttt{true} \text{ where } t = \texttt{taint(literal(false))}$$

We are now able to illustrate the coherence problem. This property asserts the attribute should evaluate to true on t, but t forwards to just `literal(false)`, where the attribute evaluates to false. As a result, this property is incoherent.

Meanwhile, it is possible there are no coherence problems with the `identity` extension, potentially resolving our earlier questions about who is to blame for the interference problem. To see why, we need to work through developing a convincing specification. Unfortunately, a convincing specification for this extension's behavior is second-order—we need to quantify over properties (which is not something we are considering in our simple definition of what a property consists of). We can imagine establishing a new property on `Expr` for `id_transform` showing something like:

$$\forall P.\ \mathit{coherent}(P) \implies P(t) \iff P(t.\texttt{id\_transform})$$

So this is a bit more complicated than would be ideal; we have to rely on the definition of coherent property in order to make this property a coherent one. (If we quantified over properties and did not constrain to just coherent ones, this would become an incoherent property!) But nevertheless, the intuition is straightforward: `id_transform` produces a tree that is equivalent in the eyes of any coherent property, and this claim can itself be stated as a coherent property (if of a more complicated sort than we want to restrict ourselves to). And so the `identity` extension is in the clear.

6.3.2 Closure properties of coherence

When exactly is a property coherent? We have already suggested the answer: we claimed that all that is necessary to coherently extend a property is to coherently extend the relations it relies upon. To show this, we will show that a property can only be incoherent by relying on an incoherent relation. This should be somewhat intuitive: a property is only incoherent if it makes some claim about t that is untrue of $t.forward$, or vice versa. The only place where we discriminate by case analysis like this (and thus could say something different depending on t) is when we're defining a relation over trees.

That incoherence sneaks in through inductively-defined relations on trees can be more precisely understood through some closure properties over logical connectives. This exercise also shows the interrelatedness of the completeness and soundness directions of coherence.

Theorem 6.3.3. *Given a coherent property $P(t)$, then $\neg P(t)$ is a coherent property.*

Proof. The completeness direction requires that $\neg P(t.forward) \implies \neg P(t)$. The contrapositive of this is simply $P(t) \implies P(t.forward)$, which we know as the soundness direction of the coherence of $P(t)$.

The proof of the soundness direction for $\neg P(t)$ is symmetric (relying on the completeness of $P(t)$). □

This shows us that the coherence of a property is preserved under negation. Notice how, under negation, completeness relies on soundness and vice-versa. Let us show that other logical connectives preserve coherence.

Theorem 6.3.4. *Given coherent properties $P(t), Q(t)$, then $P(t) \implies Q(t)$ is a coherent property.*

Proof. We again show one direction (this time, soundness), as the other is symmetric. The goal is to show that $(P(t) \implies Q(t)) \implies (P(t.forward) \implies Q(t.forward))$. Unfolding definitions, we are given $P(t) \implies Q(t)$ and $P(t.forward)$, and the goal is to show $Q(t.forward)$. We apply the soundness of $Q(t)$, leaving us to show $Q(t)$. We apply our first given, leaving us to show $P(t)$. We now apply the completeness of $P(t)$, with our final goal of showing $P(t.forward)$, which is our second given. □

Theorem 6.3.5. *Given coherent $P(t), Q(t)$:*

$$P(t) \wedge Q(t) \text{ is coherent and } P(t) \vee Q(t) \text{ is coherent} \tag{6.1}$$

Proof. These are even simpler. In each case, the soundness of the whole depends on the soundness of the parts. Likewise for completeness. □

6.3.3 Restricted propositions and relations (Limitations and scope of this development)

The next sensible step is to show that coherence is preserved by more logical constructs, such as quantifiers. However, we're stymied by our choice to talk specifically about properties. After all, if $P(t)$ is a coherent property, then what is $\forall t.P(t)$? We've removed the free variable, and so it is no longer a property.

We choose to exclusively talk about properties because this considerably simplifies the formal development we must make, and emphasizes the critical aspect of coherence. To complete this development, it would be necessary to fully generalize to coherent propositions (not merely properties). However, we are going to take a different approach. Instead of generalizing coherence to accommodate any form of

proposition, we're going to restrict the range of propositions we allow to just those that we can treat as simple properties. We believe the notion of coherence can be extended to propositions in general (and so this restriction could be lifted), but we leave this for future work.

Definition 6.3.6 (Coherent (restricted) propositions)**.** A proposition is restricted and coherent if:

1. (Restricted:) It has the shape $\forall \overline{x}.\ P(\overline{x})$, where P does not contain any quantifiers.

2. (Coherent:) For each variable x_i of tree type, P is coherent in that variable. That is,

$$\forall \overline{x}.\ P(..., x_i, ...) \iff P(..., x_i.forward, ...)$$

One liberalization we incorporate into the above definition is to allow multiple leading universal quantifiers. Our notion of a coherent property can easily generalize to multiple variables of this sort, as shown in the second rule. We simply require that any tree typed variable must be independently coherent in P. That is, we must have both $P(x, y) \iff P(x.forward, y)$ and $P(x, y) \iff P(x, y.forward)$. (As a straightforward consequence, we also have $P(x, y) \iff P(x.forward, y.forward)$.) Allowing multiple variables (in this way, where only leading universal quantifiers are permitted) poses no problem for extending our existing closure theorems to cover multiple variables as well.

When is a proposition of this form coherent? As it turns out, thanks to our closure properties, whenever the relations it relies upon are coherent. Let us first define our (also restricted) notion of coherent relation, and then show this.

Definition 6.3.7 (Coherent (restricted) relations)**.** A relation $R(\overline{x})$ is restricted and coherent if:

1. (Coherent:) For each parameter x_i of tree type, R is coherent in that variable. That is,

$$\forall \overline{x}.\ R(..., x_i, ...) \iff R(..., x_i.forward, ...)$$

2. (Restricted:) Each constructor of the relation has the form of a coherent (restricted) proposition.

Restriction 2 merely applies our restriction on the shape of propositions down through relations, and not just the initial proposition. It is restriction 1 that we rely on for our propositions to be coherent. We can coherently extend n-ary relations in the same way we coherently extended unary relations, just applying the same operation to each quantified variable in turn. (If this is not clear, we will see an example with an equality relation in the next section.)

Theorem 6.3.8 (Coherent relations imply coherent (restricted) propositions)**.** *If every relation R in a proposition P is coherent, and P contains no quantifiers, then the proposition $\forall \overline{x}.\ P(\overline{x})$ is a coherent proposition.*

Proof. The given proposition has the required form (satisfying requirement 1 in the definition of a coherent restricted proposition), and so we turn to requirement 2: ensuring that every quantified variable is coherent. We can show this by induction on the structure of the formula. Thanks to our closure properties, we can handle the cases of logical connectives. The base cases, then, are atomic formulas. Each atomic formula either does not involve any tree typed variables (as so is trivially coherent)

or is the use of a relation R, which we have assumed to be coherent. As a result, the proposition is coherent. □

This theorem focuses our attention when looking for coherence violations. It is when we introduce relations (where we do case analysis on trees) with cases for forwarding productions that we may introduce incoherence. (Equivalently, it is when we write synthesized equations on forwarding productions, which produce evaluation relations.) Beyond that, we can generally content ourselves to assume our propositions are just fine. (Assuming, that is, they are suitably restricted, but as noted, we conjecture this restriction can be lifted.)

There is one last subtlety that merits attention. Because we have restricted the shape of the propositions we can state, this can have subtle effects on other claims. For instance, consider the claim:

$$(\forall t \in H.\ P(t)) \implies (\forall t \in H \triangleleft E.\ P(t))$$

We see a restricted proposition appears on each side of this implication. But consider the negation of these propositions (something we currently do not permit, the proposition must begin with its quantifiers to meet the definition of a restricted proposition, but this restriction is something we believe can be lifted, so the following is something we'd still expect to be coherent):

$$(\neg\forall t \in H.\ P(t)) \implies (\neg\forall t \in H \triangleleft E.\ P(t))$$

This is the contrapositive of:

$$(\forall t \in H \triangleleft E.\ P(t)) \implies (\forall t \in H.\ P(t))$$

Which now looks like very different claim! (We were extending proofs from host to extension, now we're constricting them from extension to host.) We'll state our

theorems involving restricted propositions using iff instead of just implication to avoid this issue. We believe this subtlety is the only way in which this has a major impact.

Finally, a clarification about notation. We will continue to state propositions in terms of one variable (as we have been doing), because this is notationally simpler and it is straight-forward to generalize to multiple variables in the form we now permit them. We have been using notation like the following: $\forall t \in L.P(t)$. Recall that we don't actually quantify over a language, instead the language is a stand-in for a (decorated) tree type restricted to productions of the stated language. We'll continue to use this notation, but note that now we're affecting the range of quantification for all variables of P.

6.3.4 The problem of equality

We have identified relations as being the primary source of incoherence. There is a particularly common relation that has serious problems of this sort: equality on trees. Equality on other types poses no problems.

Consider how we might define an equality relation on a tree type. A term $p(\overline{x}) = q(\overline{y})$ exactly when $p = q$ and each $x_i = y_i$. But this is too strict: a tree rooted in a production can only be equal to another tree rooted in the same production! If a tree is not "equal" to what it forwards to, then equality is obviously incoherent.

Fortunately, this is no real trouble, as we can define coherent equality on a nonterminal type N easily enough. This essentially corresponds to writing down an equality relation over non-forwarding productions, and relying on coherent extension to fill in the remaining cases:

Definition 6.3.9 (Coherent equality)**.**

$$x =_N y = \begin{cases} x_i =_M y_i & \text{if } x = p(\overline{x}) \text{ and } y = p(\overline{y}) \text{ where } x_i, y_i : M \\ x.forward =_N y & \text{if } x \text{ is rooted in a forwarding production} \\ x =_N y.forward & \text{if } y \text{ is rooted in a forwarding production} \end{cases}$$

With coherent equality, a tree is considered equal to the tree it forwards to. This is accomplished by looking through forwarding productions to compare equality on what they forward to. Incidentally, this is the same definition of equality we'd get automatically using `case` expressions in AG to define the usual Boolean equality function for trees. If pattern matching failed on a forwarding production, we'd look through forwarding and try again. We will assume that every use of equality on trees, from here on, is actually coherent equality.

The above definition of coherent equality is actually not the only notion of equality we might want. Recall that we're working with decorated trees, not undecorated ones. Equality on decorated trees may also require that inherited attributes supplied also be (coherently) equal. Equality of inherited attributes can be trickier, because now we also may have to worry about equality of undecorated trees[3].

But coherent equality as we have defined it is a minimal building block for more involved notions of equality. If we care about equality of inherited attributes, we only need to check these at the root, since all lower inherited attribute get computed from there. If we only care about equality of synthesized attributes, again, we just need to check them at the root. Coherent equality is the only recursive operation deep

[3]Undecorated trees do not yet have inherited attributes, and so cannot evaluate forwarding equations. This causes no problems for reasoning because we can just universally quantify over all possible values given as inherited attributes. But it could be a problem in the object language (e.g. with test cases), where to evaluate the equation we'd need to invent a specific value, somehow.

into the tree. And so, if the inherited attributes matter, we expect these to be stated separately, and otherwise we treat equality as being just coherent equality as defined above.

Coherent equality leads directly to another subtlety, however, once again related to our quantification over decorated trees. If our goal is to construct a decorated tree, we cannot just apply a production to its children, we also need to supply its inherited attributes. Likewise, it is a type error to apply a production (that expects an undecorated child argument) to a decorated tree. Fortunately, it is straight-forward to work around these problems. Instead of directly constructing a new tree, quantify over all trees and restrict to just those with the desired shape, using coherent equality.

As an example, consider our earlier simple proposition:

$$\forall t.\ \texttt{taint}(t).\texttt{is\_tainted} = \texttt{true}$$

This runs afoul of the the above problem: we obviously want a decorated tree to access `is_tainted` on, but `taint` is just a production, which takes an undecorated tree and produced an undecorated tree. We can rewrite this as:

$$\forall t_1, t_2.\ t_1 =_{Expr} \texttt{taint}(t_2) \implies t_1.\texttt{is\_tainted} = \texttt{true}$$

Now we have a decorated tree t_1 which is coherently equal to a shape rooted with `taint`. Notice how this expands the range of what we might be accessing `is_tainted` on. It's no longer obviously rooted in `taint` by construction, but anything that might be coherently equal to it. (We don't require anything about the inherited attributes here, as that doesn't matter for this proposition.)

6.3.5 Non-interference: coherence over extended languages

With this development of coherence out of the way, let us return to the idea of non-interference.

When we say that a property P over a language H is coherent, we are making a precise claim about that property. We also give a precise mechanism for extending a coherent property to encompass new forwarding productions, without compromising its coherence. However, the consequences can be confusing if we're not careful.

Let us start by defining *noninterfering*, our modular restriction on extensions (from section 6.2.1). (We will show this definition meets the requirements of our previous definition in the next subsection.)

Definition 6.3.10 (Non-interference)**.** A language extension is defined to be non-interfering ($noninterfering(H \triangleleft E)$) if:

1. The extension is coherent. That is, each property in its specification is a coherent property:

$$\forall P \in \mathscr{P}(E).\ \forall t \in H \triangleleft E.\ P(t) \iff P(t.forward)$$

2. The extension preserves all coherent properties. Or in other words, any true coherent property about the host is a true coherent property about the extended language (and vice versa):

$$\forall P.\ \big(\forall t \in H.\ P(t) \iff P(t.forward)\big) \implies \big(\forall t \in H.\ P(t)\big) \iff \big(\forall t \in H \triangleleft E.\ P(t)\big)$$

The interesting source of confusion is this: in practice, these two requirements often look equivalent. As we'll see a little later, the way an extension can fail requirement 2 also creates incoherent properties about its behavior. However, these two requirements are very different ways of looking at the problem, and they are actually both necessary.

If we think purely about perspective 1, then the `is_tainted` extension's correctness depends on an incoherent property. The value of the attribute `is_tainted` differs between `taint` and what it forwards to, and so any claim about the attribute on this production would necessarily be incoherent. Interference arises because we will be unable to preserve this incoherent property.

If we think purely about perspective 2, then the `is_tainted` extension does not preserve coherent properties about the host language. We could have stated a coherent property that the `is_tainted` attribute would always evaluate to false[4]. This can be proven when restricted to productions from H, but of course this is false when ranging over $H \triangleleft E_A$ where `taint` can make it evaluate to true. Interference arises because coherent properties other extensions may depend upon are violated. But, as this example shows, `is_tainted` ends up violating both requirements.

It's certainly possible to violate only one of the requirements. The developer of an interfering extension might simply leave out a true incoherent property (and thus must be caught by requirement 2). (For example, consider if the `is_tainted` extension stated no properties about its behavior.) Or the developer might state an "unreasonably strong" true incoherent property (and thus must be caught by requirement 1). (For example, consider if the `identity` extension had claimed using

[4]Recall our early note about the formal meaning of quantification over trees: here is one case where we need to show a property involving this extension's attribute, but about trees consisting only of productions from H.

strict (incoherent) equality that `t.id_transform = t`.) So the requirements are not actually equivalent, they are just often violated together in practice.

6.3.6 Coherence assures modular non-interference

With that definition of *noninterfering*, let us see how it achieves our goal from the definition we gave for modular non-interference (6.2.1). The definition of *noninterfering* means we can extend coherent properties from the host to any extension (this is just part 2 of the definition). Let us work up from there.

First, observe that by our language composition operator semantics for Ag:

$$H \triangleleft (E_1 \uplus_{\emptyset} E_2) = (H \triangleleft E_1) \triangleleft E_2 = (H \triangleleft E_2) \triangleleft E_1$$

This equivalence is an important part of how we achieve non-interference. Consider how we extend properties about H to $H \triangleleft E_1$: this is straightforward from *noninterfering*. But now consider those same properties from $H \triangleleft E_1$ to $(H \triangleleft E_1) \triangleleft E_2$. We must confront the fact that E_2 was not defined as an extension of $H \triangleleft E_1$ but just H.

Fortunately, our semantics for Ag and our definitions of coherent extension give us a safe way to do this. The Ag semantics care only about what set of modules are included; the attribute grammar's behavior is never different for any "order" or shape of how they get included, so there is no difficulty there. This is because the only possible side-effect of composing Ag modules is the generation of copy equations in forwarding productions, and the copy equation for an attribute in a production will be identical, regardless of the order in which they are created. Likewise for our coherent extension of the logical specification: coherent extension only ever indicates a relation on a forwarding production should be equivalent to what it forwards to, so

no different order of composition could come to a different conclusion. This means the process for taking properties from H to $H \triangleleft E_1$ is exactly the same as for taking properties from $H \triangleleft E_1$ to $(H \triangleleft E_2) \triangleleft E_1$. In both cases, there are simply some potential new forwarding productions, which we must coherent extend relations over. Effectively we are able to "rebase" an extension onto an extended language, in the same manner and with the same properties as we extend a language. That the above is possible may be surprising, and so we wish to provide more concrete intuition.

Theorem 6.3.11 (Coherent extension of proofs)**.** *Given a proof of a coherent property* $\forall t \in H.\ P(t)$*, and given a noninterfering* E*, then* $\forall t \in H \triangleleft E.\ P(t)$*.*

Proof. While this is a direct consequence of the definition of *noninterfering*, we wish to show it in a slightly more constructive way, to build intuition. Take our given proof over H (which we assume proceeds by induction on decorated trees) and remove all cases for forwarding productions. We can then uniformly prove the subgoals $P(t)$ for each forwarding production in the same way, thus extending it to range over $H \triangleleft E$. The originating module of the forwarding production meets *noninterfering*, and so requirement 2 ensures each forwarding production preserves all coherent properties. Since $P(t)$ is coherent property, we can apply completeness to change the goal to showing $P(t.forward)$. Because we're proceeding by induction on decorated trees, this goal is discharged by our induction hypothesis. □

This gives us a procedure for *how* it is that properties can be preserved by extensions. And with that in mind, it becomes clear how we are able to "rebase" extensions, including their proofs. We simply restrict proofs to just the cases of non-forwarding productions, which extensions can never alter, and then uniformly extend them over any forwarding production, regardless of origin. This procedure isn't affected by

whether we're extending a language (going from H to $H \triangleleft E_1$) or rebasing a language extension (going from $H \triangleleft E_1$ to $(H \triangleleft E_2) \triangleleft E_1$). All these operations can do is add more forwarding productions, which we can safely handle so long as all extensions are non-interfering. And so we can do either.

Theorem 6.3.12 (Non-interference of two extensions)**.** *Given H, E_1, and E_2 such that*

- *All are noninterfering,*
- $\forall P \in \mathscr{P}(H).\ \forall t \in H.\ P(t)$,
- $\forall P \in \mathscr{P}(E_1).\ \forall t \in H \triangleleft E_1.\ P(t)$, *and*
- $\forall P \in \mathscr{P}(E_2).\ \forall t \in H \triangleleft E_2.\ P(t)$

Then $\forall P \in \mathscr{P}(H) \cup \mathscr{P}(E_1) \cup \mathscr{P}(E_2).\ \forall t \in H \triangleleft (E_1 \uplus_{\emptyset} E_2).\ P(t)$.

Proof. For properties from H, we can extend them over E_1 by the definition of *noninterfering* on each of these. For properties from E_1, we can extend them over E_2 by observing the equivalence of "rebasing" described above, and the definition of *noninterfering*. For properties from E_2, we can apply a symmetric operation to extend them over E_1. □

And, as a straightforward consequence, we can now state our definition of modular non-interference as a theorem about *noninterfering*:

Theorem 6.3.13 (Non-interference of multiple extensions)**.**

$$\begin{aligned}&\big(\forall i \in [1,n].\ \mathit{noninterfering}(H \triangleleft E_i)\big) \implies \\ &\qquad \forall i \in [1,n], P^i_j \in \mathscr{P}(E_i).\ (\forall t \in H \triangleleft E_i.\ P^i_j(t)) \implies \\ &\qquad\qquad \forall t \in H \triangleleft \{E_1 \uplus_\emptyset \cdots \uplus_\emptyset E_n\}.\ P^i_j(t)\end{aligned}$$

Proof. We recursively compose each extension together in turn, applying the above theorem. That is, we start by composing E_1 and E_2, then we proceed by composing $E_1 \uplus_\emptyset E_2$ and E_3, and so on. □

6.4 Showing non-interference

The definition of *noninterfering* requires two properties hold of an extension. The first requires that our extension's verification properties are all coherent. The second requires that our extension does not violate any coherent properties about the host (known or unknown). Let us work through this process for our running example, beginning with the first requirement.

In figure 6.5, we return to the small extension (from figure 6.4) for the `RepMin` example we showed earlier (from figure 6.3.) On the left of this figure, we re-state the production that is newly introduced as part of this extension. On the right side of this figure, we show the automatically derived extensions to the correctness relations, along with restating the intended ones (from the original figure.)

As part of showing this extension is non-interfering, we must show that our intended and inferred cases for these relations are equivalent. (Recall that writing explicit properties for forwarding productions is still useful: we're specifying the meaning of the forwarding productions, and then verifying our code meets these

```
production three
l::RepMin ::=
  x::RepMin  y::RepMin  z::RepMin
{
  l.min = min(min(x.min, y.min),
              z.min);
  forwards to
    inner(inner(x, y), z);
}
```

Inferred extensions to correctness relations:

$\texttt{lte\_all}(three(x, y, z), m)$:-
$\quad\texttt{lte\_all}(inner(inner(x, y), z), m)$
$\texttt{exists}(three(x, y, z), v)$:-
$\quad\texttt{exists}(inner(inner(x, y), z), v)$

Intended extensions to correctness relations:

$\texttt{lte\_all}(three(x, y, z), m)$:- $\texttt{lte\_all}(x, m) \wedge$
$\quad\texttt{lte\_all}(y, m) \wedge \texttt{lte\_all}(z, m)$
$\texttt{exists}(three(x, y, z), v)$:- $\texttt{exists}(x, v) \vee$
$\quad\texttt{exists}(y, v) \vee \texttt{exists}(z, v)$

Figure 6.5: Important parts of showing non-interference of the extension from figure 6.4.

specifications.) In other words, we must show these properties (as we have chosen to amend them) are coherent. Showing coherence requires a manual proof be written by the extension author. In this case, however, this proof is quite simple. For example, the inferred case $\texttt{lte\_all}(inner(inner(x, y), z), m)$ expands in a syntax directed way into $\texttt{lte\_all}(x, m) \wedge \texttt{lte\_all}(y, m) \wedge \texttt{lte\_all}(z, m)$, which is exactly what we intended for `three`. Again, we show this more formally in appendix A.

Once we have shown these relations to be coherent, we should then use them to show that our properties are coherent. Because properties are coherent if their relations are, this should be an easy task... However, we have only shown above that the defined inductive relations are coherent, we have not done so of the evaluation relations for synthesized attributes, which come from equations of the attribute grammar. We must not forget all the relations, not just those new ones we wrote defined as part of the verification. In this case, the evaluation relations are coherent as well, and so after showing that, showing the properties are coherent is immediate (and shown in the appendix).

By showing that the stated properties are coherent, we have done half of what we

need to do justify this extension as non-interfering. The remaining half is showing that the extension preserves all coherent properties. In general, it's not necessary to concern ourselves with extending the proofs of properties we've inherited from the host language. Instead, we need to show that the extension preserves all coherent properties, not just those we have in mind. The ones we have in mind will of course be included as a consequence, since we've shown them to still be coherent as our first step.

But this is a difficult problem: how can we be sure any arbitrary coherent property is preserved when we introduce a new forwarding production?

6.4.1 An "unreasonable" approach to enforcement

On the other hand, how could an extension possibly not preserve a coherent property? Again we tread close to the source of confusion we identified earlier: the almost-equivalence of the two requirements of non-interference. After all, if we've shown that a property is coherent over H, and we coherently extend it over $H \triangleleft E$... shouldn't it necessarily be preserved? Isn't that a consequence of being coherent?

Here is the subtlety: if we take a coherent property about H and use coherent extension of its relations to obtain a property about $H \triangleleft E$, we do not necessarily have a coherent property about $H \triangleleft E$. The missing piece is the *evaluation relations* for synthesized attributes, that arise from the code of the attribute grammar. Coherent extension of relations only applies to relations defined as part of the logical specification, not to the evaluation relations generated by the attribute grammar. Coherent extension of a property only preserves coherence if all relations used are still coherent. When the property evaluates synthesized attributes, the property depends on relations that may become incoherent with a new extension. This suggests a solution:

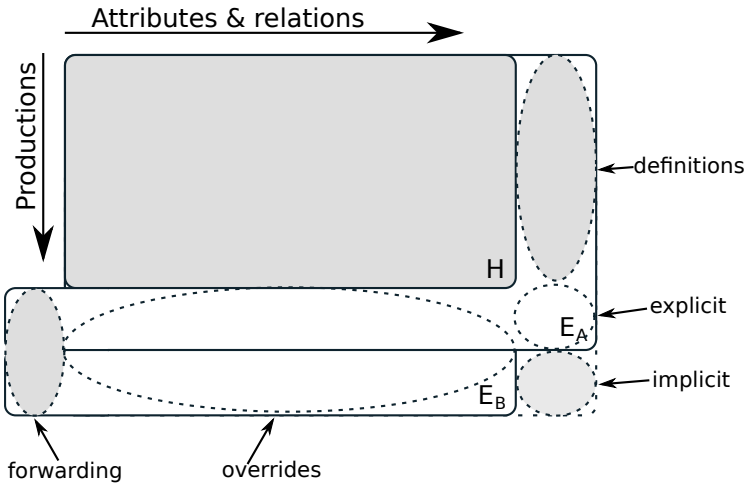


Figure 6.6: A table showing synthesized equations and relations on one axis, and productions on the other. E_A introduce both a new production and new attributes, while E_B introduces only a new production.

we can ensure the evaluation relations are coherent.

In figure 6.6, we can see a visual depiction of the potential space of interference with two language extensions shown. For the purposes of this diagram, assume there are no forwarding productions within H. We also assume that all pattern matching has been translated to attributes (and so patterns that match forwarding productions are equivalent to equations on forwarding productions). This figure depicts an extension E_A introducing a new forwarding production and synthesized attribute, while E_B introduces a new forwarding production only. The shaded regions are authoritative. This includes the host (labeled H), the definitions of new attributes ("definitions") on just non-forwarding productions (where incoherence cannot possible arise), and forwarding equations ("forwarding"). Implicit copy equations (labeled "implicit") also obviously preserve coherence.

All of our problems really stem from the other areas: labeled "overrides" and

"explicit." These explicit equations on forwarding productions are a potential source of incoherence in evaluation relations. This incoherence source may cause the extension to be unable to preserve the coherence of properties, which also means the extension may not preserve coherent properties of the host. Likewise, properties extension developers state about their extensions may be incoherent simply because of these equations.

As a result, it is actually (mostly) possible to enforce non-interference syntactically. If we ban explicit equations on forwarding productions, then the "implicit" region grows to cover that whole space, leaving no possible source of interference. Or, more formally:

Theorem 6.4.1. *If there are no explicit equations on forwarding productions, then all evaluation relations are coherent.*

Proof. An evaluation relation (we write them as $t.s = x$) is coherent when $t.s = t.forward.s$, which is exactly the implicit copy equation forwarding introduces when no explicit equation is present. □

Theorem 6.4.2 (Coherent evaluation implies preservation of coherent properties)**.** *If all evaluation relations are coherent, then the extension preserves all coherent properties.*

Proof. Suppose it does not, and there is some P which is not preserved. It must occur in the direction showing some P holds on H but not $H \triangleleft E$, since for us the other direction is just a subset (note: this is not true if we generalize beyond just our notion of restricted propositions, but it looks like a symmetric argument might hold there as well).

Non-preservation in this direction means there is some $t \in H \triangleleft E$ such that $P(t.forward)$ but not $P(t)$, where t is rooted in a forwarding production. Thus, there must be some relation R in P such that $R(t.forward)$ but not $R(t)$ or vice versa. But all inductively defined R are coherent extensions of coherent relations, and the evaluation relations of synthesized attributes are also assumed to be coherent. Therefore, there can be no such R, contradicting our assumption. And so we can conclude that the extension does preserve all coherent properties. □

The above is proof of requirement 2 of our definition of non-interference. Banning explicit equations entirely is sufficient to ensure we can preserve all properties. (But not necessary! The breaking up of the above theorem in two pieces with "coherent evaluation relations" in the middle foreshadows development we will make in the next section.)

What about requirement 1? Unfortunately, it is always possible to write an incoherent property for even the most well-behaved extension. Consider the `identity` extension which we found to be potentially coherent. We could have instead tried to make a promise with this extension that we could not deliver on. For instance, asserting that `t = t.id_transform`, using an incoherent notion of tree equality[5]. So this restriction does not exempt extension developers from having to ensure requirement 1 isn't violated.

We can, however, have quite a lot of confidence that the intended properties are coherent in practice. Every test case style of property (that is, the propositional claim that a Boolean AG expression evaluates to true) is coherent. Since these are all

[5]Actually, this is difficult to do, as without equations on forwarding productions this exact property would be false for the `identity` production. One could try to exempt this production, but then our property is becoming more contrived. But contrived or not, such properties exist.

preserved by this extension, we could not prove any incoherent property that would contradict them. This significantly restricts the scope of what kinds of incoherent properties we could try to write. We can generalize this claim somewhat.

Definition 6.4.3 (Observable propositions)**.** An observable proposition is one that makes use of no relations over trees except coherent equality and Ag attribute evaluation relations.

Theorem 6.4.4 (Observable propositions are coherent)**.** *If every evaluation relation is coherent, then every observable proposition is coherent.*

Proof. Since propositions are made incoherent by making use of incoherent relations, and every relation involving trees is either coherent equality or an evaluation relation which we have assumed to be coherent, then the proposition must be coherent. □

Observable propositions are a fairly narrow class, but even so they are quite useful. For `taint`, we might wish to state a proposition such as

$$\forall t.\ \mathit{tainted}(t) \iff t\texttt{.is\_tainted = true}$$

This is no longer an observable proposition because it involves defining a relation over trees like *tainted*. However, it is interesting to note that *tainted* is pretty much just identifying a subset of trees for us. Every valid instantiation of this proposition with a specific tree gives us an observable proposition. This generalizes, but for rather trivial reasons: the reason we can get into incoherence trouble is because quantification over trees may include trees that were originally inconceivable, and so we could not have written down such an instantiation.

And so, this syntactic restriction can't quite automatically ensure the extension will be non-interfering, but for the rather small reason that the extension may be

explicitly promising behavior over unknown other extensions that it cannot deliver on. In our experience so far, we've never seen this happen. This is not just because we do not verify extensions in practice, but because this is an unusual sort of property to have in mind, even merely as documentation of the extension's behavior. It is a property for which one cannot write a test case nor give any concrete example. To state it, one must admit the possibility of composition with unknown other extensions, and then explicitly say that the extension will behave in a way that one cannot be sure of.

As a result, although this unreasonable approach does not guarantee that extensions meet requirement 1 of *noninterfering*, it still provides some confidence, even without having done any verification. And we can be certain that it meets requirement 2.

6.4.2 Relationship to macro systems

This approach (syntactically forbidding explicit equations on forwarding productions) is nearly that of macro systems, and this gives us the chance to consider the relationship between our development with attribute grammars and other approaches to language extension.

Macro systems that do not permit case analysis of subtrees clearly enforce non-interference, though this is much more restrictive that what we suggest here. If you cannot analyze subtrees, then this is equivalent to not only banning explicit equations on forwarding productions, but also forbidding accessing synthesized attributes on children in forwarding equations as well.

When subtree analysis is permitted, it is still possible to do so in a non-interfering way. Bottom-up expansion of macros, for instance, would mean extensions (in the

form of macros) are non-interfering because subtree analysis could only ever examine host language trees (all macros in subtrees would already be rewritten away).

With a top-down evaluation order, it is possible for a disciplined approach to lead to non-interference. Whenever a macro attempts to pattern match on a subtree that is rooted in another macro, it must always expand that macro and then resume pattern matching on the result instead, to ensure that it doesn't "get stuck" looking at syntax it doesn't understand. So long as macro evaluation does not have side-effects, this is safe, and it almost functions like pattern matching looking through forwarding (except that forwarding preserves the original production, instead of being rewritten away).

However, no macro system we are aware of enforces any of these disciplines. In practice, many macro systems use top-down evaluation order, in part for lack of grammar: the system does not know what pieces of a macro invocation's arguments might be host language constructs, within which we might looks for more macros. The lack of grammar makes it difficult to be disciplined, as was suggested above: where might another unknown macro appear, so that we should anticipate evaluating it before matching on the result? (Indeed, the answer could be "almost anywhere.") Worse, as a result of this reliable top-down order, some macros are written to *expect* to be able to match subtrees and find references to new symbols or to other macros, to special case their behavior[6]. This leads to interference problems, and it's exceptionally easy to break these systems by trying to introduce additional macros.

Term-rewriting systems can also manage to be non-interfering through the exclu-

[6]As an interesting anecdote, we searched for a long time for a way to achieve non-interference while preserving this sort of arbitrary incoherent subtree analysis capability of existing macro systems. It wasn't until we reconsidered and thought to disallow this that a means of ensuring non-interference occurred to us. In retrospect, we now consider this to be a serious flaw in how macro systems have traditionally been designed.

sive use of bottom-up evaluation order. This requires additional discipline, to ensure than all extension productions are introduced along with a rewrite rule that translates them into the host language—similar to the forwarding equation in our system, or to how macros are expanded.

6.5 Attribute properties: more useful non-interference

In the previous section, we banned explicit equations on forwarding productions, in order to ensure evaluation relations were coherent. However, we may need to place some equations on forwarding productions, which is why we call this approach "unreasonable." As a straightforward relaxation, we can allow equations but instead require that $t.s = t.forward.s$ hold for every s, but so far it looks pointless to do so: this is exactly the implicit copy equation forwarding introduces. Alone this is no different than the "unreasonable" approach. We will now turn our attention to relaxing this burden, to permit differences in the values of attributes between host and extension.

6.5.1 Host language adaptation

One of the major weaknesses of the "unreasonable" approach is that it limits our ability to make extensions "feel" like part of the host language. To be non-interfering, it must be the case that all attribute values are equal between forwarding and forwarded-to trees, including attributes like an `errors` list. This means, for example, that we are unable to raise any error messages about an extension's custom syntax. The errors that can be raised are only those which the host language raises on the forwarded-to tree, which may be about generated code that does not appear in the program the

```
production errorExpr
e::Expr ::= msg::[Message]
{
  e.errors = msg;
}
```

```
production bridge
e::Expr ::= x::ExtensionAST
{  forwards to if null(x.errors)
     then x.translation
     else errorExpr(x.errors);
}
```

Figure 6.7: An error production, and a typical example of its use.

user wrote.

One way to mitigate this problem is to make a minor change to the host language implementation. The `errors` attribute must always be the same, but if we have host language abstract syntax capable of raising arbitrary error messages, this would be a non-issue. To do this, we introduce into the host language *error productions*, like that in figure 6.7, that simply always raise the error messages they are provided with. Forwarding productions in extensions, instead of trying to override the value of the `errors` attribute (which is incoherent), can instead simply choose to forward to the error production when they needs to raise custom errors. We still demand strict equality on the part of the `errors` attribute, but it's no longer a problem.

However, there are other important ways to make an extension "feel native" beyond just error messages, such as attributes giving the "type representation" or `typerep` of expressions. (Which, incidentally, also matters for the construction of good quality error messages.) If we are introducing an extension type, we will presumably want syntax that constructs this type. However, strict equality would require all expressions to only yield types that the host language could yield, which means only host-language types and not the extension type. This would make having an extension type essentially useless, and is another reason why the "unreasonable" approach is insufficient.

$$e.\texttt{host} = \begin{cases} e & \text{if } e \text{ is a terminal} \\ p(\overline{c.\texttt{host}}) & \text{if } e = p(\overline{c}) \text{ where } p \text{ is non-forwarding} \\ e.forward.\texttt{host} & \text{if } e \text{ is rooted in a forwarding production} \end{cases}$$

Figure 6.8: A transformation eliminating forwarding productions.

6.5.2 Weakening the strict equality condition

Copy equations aren't the only equations that ensure we preserve all coherent properties, thankfully. For attributes of non-tree types (e.g. `Integer`), $t.s = t.forward.s$ is a coherent property, and so the copy equation is the best we can do. But for tree-type (i.e. decorated or undecorated nonterminal type) synthesized attributes, we previously observed that strict equality is incoherent. Examples of these attributes (on a hypothetical language) include `typerep` on expressions (which gives their type) or `defs` on declarations (which gives a list of definitions to introduce into the environment). We can now leverage the incoherence of strict equality on trees into an advantage.

We begin with a useful tree transformation. In figure 6.8, we define a transformation `host` that eliminates all forwarding productions from a tree. This transformation has a very useful property:

Theorem 6.5.1. *Given any coherent property $P(t)$, $\forall t.\ P(t) \iff P(t.host)$.*

Proof. By induction on the decorated tree t. When t is terminal, $t.host = t$ and the goal is trivial. When t is a non-forwarding production, we need only appeal to the induction hypotheses for all children. When t is a forwarding production, we appeal to the coherence of P, the induction hypothesis about $t.forward$, and finally the

observation that $t.forward.host = t.host$. □

In other words, this transformation preserves all coherent properties. With this tool, we can observe a useful property about the evaluation relation (R) for a tree-type synthesized attribute s:

$$R(t, x.host) \iff R(t, x) \iff R(t.forward, x) \iff R(t.forward, x.host)$$

The outer two iffs are a result of this theorem, while the inner one is a result of coherence. As a result, if $t.s.host = t.forward.s.host$, this is enough to know that the evaluation relation is coherent.

This means that instead of explicit equations being pointless (because we need $t.s = t.forward.s$ to preserve coherence of evaluation), we actually can have a meaningfully different value computed for a forwarding production compared to what it forwards to. We have found a way to allow some kinds of useful explicit equations. Despite the differing value, we still have a coherent evaluation relation, and so we know that we're still preserving coherent properties, and any observable property of this language is still coherent.

This justifies variations in the values of attributes between a forwarding production and the tree it forwards to. Instead, it must only be the case that what these values themselves eventually forward to is equivalent. In a similar way, we can allow an extension expression to have a `typerep` that is an extension type, so long as those types are eventually in agreement. Often, this will have the form $t.typerep.forward = t.forward.typerep$, essentially requiring these operators commute.

However, this weakening of the strict equality rule, though seemingly very powerful, is actually still essentially useless without further liberalization. Although we

can have differing values between forwarding and forwarded-to trees, these values are themselves trees. And on those trees every attribute yielding a non-tree type (e.g. string, integer, or some other terminal type) must still be strictly equal. And these non-tree values are the only real way we have to "observe" a difference between two trees. Our extension type went from being useless because no expression production could have it as its type, to essentially useless because it cannot be *observably* different from the type it forwards to. But as soon as there's any useful way to observe differences at all, this technique becomes very valuable.

6.5.3 Closing the world

We've required that every evaluation relation be coherent and thus preserve all possible coherent properties. This is a good default assumption: we can never know what other property some extension might someday want to rely on. But what if there really is only a few coherent properties we could ever want to know about a specific attribute?

Let us consider the example of "pretty-print," the `pp` attribute. Consider a property like $t.pp.parse.host = t.host$. This property is essentially the claim that pretty printing composed with parsing is equivalent to the identity function (once again using `host` in order to make this modulo forwarding). That's essentially all we need to know about a pretty printing, extensions don't really need to introduce new properties for this attribute.

So we can have the host language specify this property, and then close the world for this attribute, so that *only* this property need be preserved. No longer must extensions preserve *all* possible coherent properties about this attribute, just this one.

We must still show this property is coherent, however. This is interesting, because we need a coherent property arising from a potentially incoherent evaluation relation. (Note that we've tried to eliminate incoherent relations to ensure we always have coherent properties, but it is possible for a property to be coherent despite using an incoherent relation, just not vice-versa.) This potentially means that every extension developer must think about whether they're preserving the coherence of this property with every equation they introduce. This also means that the implicit copy equations of forwarding must always be an acceptable value for these attributes. Even if a forwarding production provides an explicit equation, it is always possible that the tree will have been replaced by what it forwards to (this is in fact our original conception of what interference might be, like the `identity` example at the start of the chapter). As a result, we only get the equation from a forwarding production on a best-effort basis, not a guarantee. So even closing the world does not excuse us from ensuring some form of "semantic equivalence."

With this change, we can now have observably different values for attributes on extension syntax versus the host language syntax they forward to. Extension developers must only prove $t.pp.parse.host = t.host$ for each forwarding production (essentially showing their explicit pretty printing equations are correct). Combined with our earlier loosening of tree-valued attributes like `typerep` and `defs`, this relaxation is quite useful. Error messages that use the pretty printing of types of subexpressions (or of variables looked up in the environment) will now be about extension types and extension syntax.

The only drawback of closing the world of properties about a specific attribute is that we must be sure our extensions do not accidentally start relying on a new property. This is unlikely to happen for `pp`, but a contrived example might look like

trying to determine if a sub-expression is an integer literal by applying a regex on its pretty-print, instead of by analyzing the abstract syntax. In practice `pp` is the only such attribute we apply this technique to, and so this has not really been a concern.

6.5.4 Summary

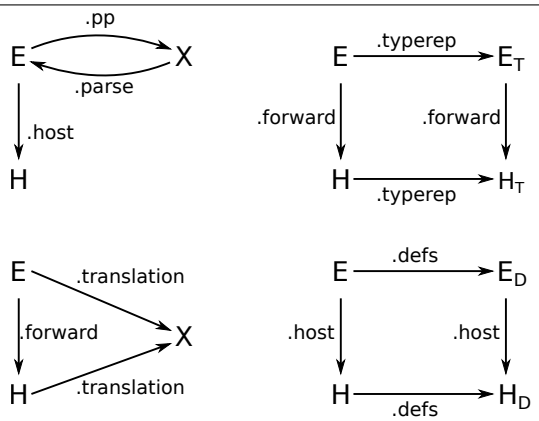


Figure 6.9: Commutative diagrams showing the relationships between forwarding productions' attribute values and their forwarded-to trees.

Many of these non-interfering properties can be described (or at least, a special case of them can be described) as a form of commutative diagram, as we show for a few of them in figure 6.9. We show pretty printing composed with parsing as actually being the identity function (which is true if every production provides its pretty print.) In the case of a hypothetical translation attribute, we show a strict equality of values between the two trees. For types of expressions, we show the special case of an extension expression and expression type, both forwarding to host language. For lists of definitions, we show a special case of commuting `host` with `defs`.

In general, we obtain this rich structure from the ability for tree-valued attributes to vary, so long as they ultimately forward to the same thing. After that, we permit

observable differences just in pretty printing. The attribute for errors must still be strictly equal, but this is accommodated just fine through error productions.

Despite the simplicity of this approach, we believe it is capable of handling real-world languages, with practical implementations. In particular, we should be able to make language extensions feel like built-in language features, without causing interference. We will see some examples of this in the next chapter, in particular a language extension that introduces a new type such as we will see with algebraic data types in section 7.3.3.

6.6 Property testing

To this point, we have freely taken a verification perspective. Now we want to take something from this development, but leave the verification aspects behind, as we consider it impractical to verify language extensions at this time. As it turns out, our approach to ensuring non-interference gives an excellent way to skip verification and attempt a testing-based approach instead. This approach is made possible by the previous section on attribute properties.

We are able to show non-interference by simply not stating any incoherent properties and by ensuring attribute evaluation relations are coherent. We showed that evaluation relations are coherent by ensuring simple equalities hold for every single attribute, which we summarized with some commutative diagrams in the previous section. We summarize the equalities used for some common attribute in figure 6.10. These equalities are easily amenable to QuickCheck[82]-style randomized property testing.

This immediately suggests the idea of doing randomized testing of these properties

```
pp       (Closed) t.pp.parse.host = t.host
errors            t.errors = t.forward.errors
typerep           t.typerep.forward = t.forward.typerep
defs              t.defs.host = t.host.defs
```

Figure 6.10: Common attributes and their proposed coherence equations.

instead of proving them. This gives us confidence that we aren't making any evaluation relations accidentally incoherent. And we showed that this would mean that our extension preserves all coherent properties, and that all observable properties are coherent. All this, without ever having to prove anything, or even having to write a specification.

We will also apply QuickCheck-style testing in a somewhat novel way. The properties we want to check apply to attributes that occur on nonterminals. For example, we might wish to check that $t.pp.parse.host = t.host$ where t are expressions. In the usual QuickCheck style, we would generate random trees t of that nonterminal type, and check that each property holds on each of those trees. Instead, while we still specify these properties in association with the nonterminal, we will emit QuickCheck-style tests on a per-production basis. Values will be randomly generated for each *child* of each production (and for the inherited attributes given to the production), and the nonterminal's properties will be checked on the resulting decorated tree rooted in that production.

This form of specifying properties on an *interface* (here, nonterminals with their associated set of attributes) and having them be checked upon all *implementations* of that interface (here, productions) is a rather interesting language extension in its own right. Applied to Haskell we could, for example, encode the Monad laws as properties on the Monad typeclass, and then have automatically generated test cases

for every instance anyone attempts to write for it. We're sure this isn't a novel idea, but we can't help emphasize that this is a good example of a language feature that seems to be made practical by language extension. Without language extension, the host language feature would have direct dependencies on something in a library like QuickCheck, which makes language designers hesitant to set in stone a design they feel they must then support forever. But that reluctance to make a commitment to a specific library design only exists for language designers, not the users, who must already make such a commitment in order to use any library at all. As a result, this sort of language feature exists in no programming language we're aware of.

An extension that works this way is able to perform some optimizations, by merging all attribute properties for a nonterminal into a single test. Each nonterminal will have multiple attributes, each with their own set of properties. But we can avoid generating separate trees for each individual property under test. Instead, when we generate trees for each production, we can test all of the properties of the nonterminal on each tree generated.

Concretely, we extend Silver in two ways. First, we develop an analog to the Haskell "deriving" clause that automatically generates a random tree rooted in a particular nonterminal. That is,

```
derive Arbitrary on Expr, Stmt, Decl;
```

This relieves us of the burden of manually writing many of these functions. This provides us with functions that can randomly generate trees of that type, with a probabilistically limited depth.

Silver already had an existing language extension for testing that allows tests to be declared at the top-level of a file. This allows one to, say, define a function and follow

it up with several test cases. When the package is built, the test cases are built along with it, and can be run if desired, or otherwise do not affect the normal use of the package in any way. The next extension we introduce augments this existing testing extension to allow us to write property tests on a per-nonterminal basis. For example, the following states the properties that should be true of the `typerep` attribute:

```
testFor t :: Expr,
  t.typerep.host == t.forward.typerep.host;
```

This extension will group these properties for each nonterminal and then, as an implicit part of each production declaration, it can emit ordinary test cases for that production, generating random trees and checking each property on each tree. This means all productions will be subject to test, even those written outside the module the property is stated within.

As a result, extension developers will have test cases checking for interference violations automatically emitted for every synthesized attribute on every production they introduce. The properties can either be explicitly specified by the host language developer (or by the extension developer, for extension nonterminals), or they can be implicitly assumed to be $t.s = t.forward.s$ absent any indication otherwise.

6.7 Application to a real compiler

As an evaluation of this approach to testing non-interference properties, we apply this property testing framework to AbleC, a specification of a C11 compiler front-end. AbleC will be described in detail in the next chapter. AbleC offers a good, though small (as we explain in a moment), natural experiment for detecting interference. It comes with several language extensions, but some of them were implemented prior to

beginning our work on non-interference. We know from inspection that one of those extensions definitely violates it. (We call this a *small* experiment because there is just one extension of this sort, not because we're calling a C front-end small.) And so we look to see if the testing approach successfully finds this violation.

This evaluation was the first time applying the random testing approach to AbleC, and so we discovered a number of bugs in the host language that needed fixing. Once those were out of the way, we spotted a few ordinary crash bugs in a couple of extensions (implementation deficiencies similar to the host language bugs we found.) Each of these were ordinary bugs, not related to interference problems. With those resolved, every extension we expected to pass in fact did so every time. For the extension we expected a failure from, the testing approach found failures 100% of the time.

Inspecting these failures, we discovered that there were, in fact, two sources of incoherence, while we had only discovered one of them beforehand. The reason we found the non-interference violations 100% of the time was that one of these violations was guaranteed to occur (the production was always adding more `defs` than the tree it forwarded to, so the lists had different length.) When we restricted ourselves to looking only for the other violation, the testing approach discovered the failure in 86 out of 100 trial runs. This was despite only generating 10 trees per production, a relatively low number for random testing, and so we consider this extremely good. This particular violation was explicitly overriding the errors attribute, rather than forwarding to an error production, and so it was occasionally hidden when the random trees turned out valid according to that particular error check.

This evaluation on AbleC is modest, however it does at least serve as a good sanity check on our intuitions. Violations of the non-interference principle we knew of were

spotted quickly by the approach, and another was discovered that we hadn't noticed initially. Both appeared to show up in random testing with very high frequency. This interfering extension had actually caused us no problems beforehand, even when it was composed and used with all other extensions we had written. However, there were indeed lurking composition bugs, waiting for the right kind of extension to come along and trigger them.

This evaluation gives us some information about how well this testing finds interference problems. But what sort of extensions are possible to write under the constraints of non-interference? We leave answering this question to the next chapter.

6.8 Related work

The notion of interference of language extensions is similar to that of feature interaction (e.g. [83]) in software product lines [84]. A primary difference is that in software product lines [85] it is typically assumed that an expert in the domain of the software is involved in the composition of various features, and can thus intervene (even though this is undesirable) if some undesired interaction is detected. This differs from our aims in which a programmer that is not an expert in language design or compiler implementation determines what extensions or features are to be composed with the host language and thus intervention by an expert is not possible. We wish to ensure there are no invalid configurations.

The testing of context free grammars [86] and attribute grammars [87] has been studied before. But this work investigates issues of test coverage and focuses on a general notion of correctness and not on non-interference of language extensions.

Instead of a testing-based approach to ensuring non-interference, formal verification of these properties is possible as well. These properties could perhaps be expressed in a dependent type system and then verified, for example building on a dependently typed attribute grammar [88]. This particular approach encodes attribute grammars in the dependently typed language Agda [89]. In this sort of framework, non-interference could be proven.

While most extensible language frameworks seems to value expressiveness over reliable composition, not all do. Wyvern [35] is the only extensible language system besides Silver, to our knowledge, that supports *reliable* composition of independently developed language extensions, at least syntactically, without abandoning parsing in favor of projectional editing. However, their approach does not accommodate introducing new analysis of the host language, and so is similar to syntactic macros in being non-interfering, but more limited.

SoundX [90] takes an interesting twist on macros by defining a desugaring over typing derivations rather than syntax trees. This gives it added power above normal macros systems: the ability to make use of type information in expansion, and the ability to define that type information for the new syntax. Like Wyvern however, this approach is limited because you cannot define new analysis ("judgements") over the host language. As such, the power of this system hovers somewhere between a macro system and our approach. It is able to achieve non-interference by ensuring that extensions are desugared before they are able to interact. Its primary contribution is a process for automatically proving that desugarings introduce no type errors, and so can be safely applied. This approach is like a special case of coherence: the type rules for the original and desugared forms must be consistent with each other.

6.9 Discussion

Coherence in the end provides us with a precise definition of what it means to forward to a "semantically equivalent" tree. We have shown that all coherent properties can be preserved by extensions, and so when extensions rely only on coherent properties, they can compose reliably. Ensuring our extensions are non-interfering places restrictions on the kinds of extensions we can make. These restrictions largely take two forms: we cannot over-promise with incoherent properties, and we cannot violate coherent properties with our extension's behavior.

Returning to our opening example of the `taint` extension, we see an example of an extension that is not possible, due to inherent interference. Note, however, that this extension is not possible *for this particular (primitive) host language*. The host language is so simple, there is no alternative implementation strategy available except interfering ones. However, this is a host language-relative restriction. An extension implementing something like `taint` is perfectly possible for even just a slightly more sophisticated language (e.g. one with functions). We will see some more examples of host language restrictions like this in the next chapter.

The attribute property testing approach we have developed is able to enforce non-interference with two caveats. First, while it ensures the code is not the source of interference problems, it cannot rule out over-promising (whether via specification or documentation). And second, the testing-based methodology (instead of verification) is obviously not guaranteed to find all problems. However, we are still able to catch obvious interference early on. So long as we do not worry about extension developers that are hostile to the idea of reliable composition, we believe it can effectively inform well-meaning extension developers of mistakes they are making. As a result, extension

developers who are still learning how to create composable extensions can be caught trying to create interfering extensions early on, before they have invested a lot of effort into something that may not be workable.

Despite these caveats, the fact that our definition of interference provides blame should ensure a healthy ecosystem. Even if some extension's developers are obstinate, the existence of blame ensures that users will be informed that this extension has serious bugs its developers refuse to fix, and users will hopefully be able to make correct decisions with this information.

As a final note, we observe that our natural language extension development methodology is unlikely to yield interference bugs. A typical language extension will start its development in a "macro-like" fashion, without explicit equations on forwarding productions. We called this "unreasonable" ultimately because it does not permit language extensions to behave like built-in language features: error messages in particular suffer. But this problem does not matter during initial development of the extension, it only matters when the extension is being "polished" for delivery to end-users. However it does mean that extensions are unlikely to rely on interfering behavior in the first place, especially with the interference tests immediately warning extension developers of their early attempts to do so. As a result, we believe this approach to avoiding interference without resorting to verification will be quite effective.

6.9.1 Future work

This development made a few critical assumptions, all of which could be relaxed by future work. First, we generally assumed that our properties are proved by induction on decorated trees. Some properties are proved by induction on other types, for

example by induction on the structure of a well-typedness relation (i.e. induction on `of term type` "the witness that `term` has type `type`" instead of on `term` or `type` themselves). It may be possible to develop a framework for reasoning about such proofs.

Likewise, we restricted propositions (and relations) to a very simplified form. We only permit universal quantification, and only out front. Generalizing the notions of "coherent proposition" and "coherent relation" to remove these restrictions would be a useful future direction. Doing so would likely require picking a particular logic, however.

With a generalized notion of coherent propositions, we'd need to re-verify the critical theorems we've shown. As we noted a few times in this chapter, we may be relying on the particular structure of properties in our proofs, and these may require work to generalize. We've tried to make note of places where it might have effect, but we cannot promise we've found them all.

Finally, there may be other approaches to achieving non-interference. Our definition of *noninterfering* is just one way of achieving this result, making use of coherence. There may be others, as we somewhat implicitly noted when making a comparison to macro systems, with unknown advantages or disadvantages.

Chapter 7

AbleC: Synthesis, Platform, and Evaluation

We have presented two modular mechanisms for ensuring that language extensions will compose successfully and without unexpected behavior. Together with the modular determinism analysis for ensuring composition of syntax, this constitutes a complete story for generating extended compilers. What remains is to evaluate whether these restrictions are too much of a burden. However, all of these analyses are in some way relative to the host language. That is, different host languages will have different "spaces" of permitted extensions surrounding them. In this chapter, we begin exploring some of the characteristics of this space.

There are a number of different questions we would like answers to. Is this space of extensions fairly large or very constrained? What kinds of extensions are possible under the constraints of these analyses? What sort of commonly desired extensions are not permitted? What host language features are important to enable broad, but still reliable, extensibility?

We cannot hope to definitively answer all of these questions. However, we wish to get at least some sense of what some of these answers may look like.

To that end, in this chapter we present AbleC, a C compiler front-end written in Silver, for which we can build reliably-composable language extensions. AbleC is our tool for exploring a single point in the space of possible host languages. But it is an especially practical point: C is in common use, and we can imagine a large number

of very useful extensions. We build a few simple extensions, following the rules we have established for building reliable composable extensions, to sample points within this space.

The central contributions of this chapter are the presentation of AbleC as a platform for future research, and the exploration of:

- What general kinds of extensions we can build for AbleC.
- A few specific extensions we have built, and some issues involved in their development.
- What kinds of extensions we cannot build for AbleC.
- What sorts of changes to C we wish we could make, to enable more and better extensions. We also attempt to generalize this into some implications that reliable language extension may have on programming language design in general.

As a general running theme, we do see some significant difficulties implementing some kinds of extensions that seem like they should be reasonable. (That is, ignoring extensions that obviously cannot be embedded into C.) However, many of these difficulties fall into two broad categories. First, many of them are simply obstacles to doing things in a particular way, and another way is perfectly possible. These turn into difficulties because the "most convenient" means of implementing an extension are ruled out, usually for potential interference. Second, many difficulties are manifestations of problems with the host language. While some of these turn out to be a problem for C, they may not be a problem for a host language designed with language extension in mind. Further, sometimes this is a problem not with C, but

with AbleC, and the implementation of the host language (the attribute grammar) can be revised.

We will begin in section 7.1 with an introduction to the AbleC compiler. Following that, in section 7.2 we will show at a high level how large classes of domain-specific languages can all be re-cast as composable language extensions. In section 7.3 we get very concrete and explore a small selection of language extensions we have implemented for AbleC, and their various design challenges. With the experience of those concrete examples in mind, we discuss in section 7.4 some general language features we are unable to build as extensions to AbleC, the lack of which is badly felt in restricting the kinds of extensions we are able to build. In section 7.5 we consider broadly how reliable language extension as a tool can better inform the host language design process. Finally, we consider some future work for AbleC (section 7.6), look at a large collection of related work in extensible languages (section 7.7), and conclude in section 7.8.

7.1 The AbleC compiler

We choose to build a C compiler, as our point of exploration of the space of permitted language extensions, in part due to the existing demand for extended variants of C. For example, the GNU C compiler has integrated OpenMP pragmas, and the Cilk [91] parallel programming features are available in the Intel C compilers. Variants of C exist for parallel computing such as the CUDA[92] and OpenCL[93] compilers. C has had several other extensible compilers implemented for it already, including xtc [94], Xoc [95], and mbeddr [96]. As such, we believe it is a good starting point.

AbleC is a C compiler front-end conforming to the C11 standard, written in Silver, and thus supporting reliably composable language extensions. Silver compiles to the JVM, and so AbleC has the potential to be reasonably cross-platform, though the current distribution is designed for use on Linux and with GCC. It currently implements all of the C11 concrete syntax and has a "enough to be useful," though not yet complete, type checker.

AbleC also incorporates a number of the GNU extensions to C. These are built directly into the host language, not as composable language extensions, because many of them are not (easily) expressible in terms of the host language. This includes a number of features that turned out to be extremely useful for language extension, such as the "statement-expression" GCC extension[1], which have important implications in what kinds of extensions AbleC allows under the composition restrictions. Sufficient other GCC extensions have been included to allow AbleC to fully parse the (glibc) C standard library and POSIX header files, all of which are included in AbleC's test suite. These extensions are not portable to all other C compilers, but several other compilers advertise GCC compatibility (as many of these appear in, for example, the Linux kernel.)

AbleC's internal structure is somewhat modeled on the structure of the LLVM C compiler, Clang. Some deviations are necessary due to the purely functional nature of our implementation, whereas Clang's internal design (and, distressingly, the C standard) very much embraces mutation. The concrete syntax is almost directly modeled on the C standard, making it easy for anyone familiar with the standard to find the right nonterminals to extend with new syntax. Most of the deviations from

[1] We'll explain this GCC extension more later, but it allows statements to appear within expressions.

that are a result of practical requirements to deal with ambiguities inherent in the C standard, such as the lexer hack. (This hack is a necessary result of the introduction of `typedefs` to the C grammar, where we must use name binding information to decide whether an identifier refers to a type or a variable, and feed this information into the lexer and parser.)

Because AbleC accepts all of plain C, it is easy for programmers to experiment with new language extensions on their existing C code bases. Switching to AbleC on an existing code base should be as simple as changing '`CC=gcc`' to '`CC=ablec`' in the makefile, and placing a shell script in their environment's path. Users can then decide on some potentially useful extensions, and evaluate the use of the new language features in their existing code, with no large up-front costs (like having to rewrite the application in an entirely new language.) An evolutionary approach such as this has been proposed before, in the Ivy language [97] (and, of course, C++), but as a new language with C compatibility instead of an extensible version of C.

7.2 Permitted classes of language extensions

We have described limitations that extensions must live under in order to ensure they can be safely composed, but these restrictions were fairly low-level in the implementation and may be difficult to map back to a higher-level understanding of what sorts of extensions are possible. We have mentioned some upper limits on the space of extensions, for instance that all extensions must somehow be denotable in the host language. Here, we turn our attention to establishing some lower limits on the space of possible extensions. We begin by observing that the *entire class* of external and embedded DSLs can be recast as language extensions that satisfy the requirements

for composition. We further observe that doing so allows these DSLs to be improved in a number of ways as well.

7.2.1 External DSLs

By an external DSL we mean a separate language, with its programs typically written in a separate file, that a tool will analyze and translate into a (for our purposes) C program to be given to a standard C compiler. The prototypical example of an external DSL is the YACC parser generator, or its modern descendants. External DSLs generally have complete freedom with syntax and analysis, but are often meant to be understood by lighter-weight tools than full compilers. For example, YACC treats semantic actions blocks–intended to contain C code to be executed on a parser reduce action–as uninterpreted strings, rather than as parsed syntax.

To turn an external DSL into a language extension, we can simply package the DSL's concrete syntax syntax as-is within the C host language using any desired form of *quoting*. For example, within a '`yacc { ... }`' block. Quoting involves using a marking token to signal an unambiguous transition to the extension syntax, and the only other imposed constraint is that the chosen ending terminal (in this example, '`}`') must of course be one which has no lexical conflicts with the extension syntax. All other potential problems in composing the syntax of these two arbitrary languages together are resolved by context-aware scanning. The rest of Copper's modular determinism analysis concerns itself with extensions that are re-using host language syntax within extension syntax, which we are not (yet) doing here, since external DSLs do not do so either. (We will see how to improve upon those "uninterpreted strings" shortly.)

On the semantics side of things, because the DSL's syntax is entirely its own (con-

taining no transitions to host language nonterminals,) the modular well-definedness analysis makes no restrictions on it. Like the Copper analysis, Silver's modular analysis comes into play when re-using host language syntax (where one may encounter other, independent extensions), and so imposes no burden on the implementation. The only change required is that the very top level production (‘`yacc { ... }`’) must forward to an abstract syntax tree in the host language. An external DSL already translates to the host language, and so the only difference is that it must now construct an abstract syntax *tree* rather than a string of concrete syntax.

To generate trees in our parser generator extension, we must deal with those "uninterpreted string" that appear as the semantic actions. It is still possible to mimic the old style: perform our string substitution and then run the parser on the resulting string to obtain a tree. But this immediately presents one possible area of improvement over an external DSL: we could replace those blobs, instead using the host language nonterminal for statements, thus re-using the host language implementation. This approach has several advantages: for example, generating error messages that are about the code the user wrote, rather than the code after it had been manipulated and emitted elsewhere (such as in a generated file.)

This improvement comes with a drawback, however. YACC has syntax within semantic action blocks to refer to children of the production. For instance, ‘`$1`‘ refers to the first child. This syntax, once we start re-using host language, essentially constitutes a separate syntactic extension (now to expressions, rather than top-level declarations.)

And now the restrictions of the modular analyses come into play: there is no direct way for us to pass information down from the enclosing yacc block, through the host language nonterminals, to the expression. This particular case can be worked

around, however. These expressions can simply look up specially-named variables (e.g. ‘`__yacc_child_1`‘), and decide to forward to a simple variable access or to an error production. In this case, we're able to "embed" the information we need into the ordinary C environment, in mostly the manner we wished. This is slightly less than ideal, but is still a significant improvement over the status quo. A more invasive change would be to change the YACC grammar slightly to actually give variable names to the children, eliminating the ‘`$1`‘ syntax entirely.

The ability to interact with the host language goes beyond just parsing. The language extension has the standard host language machinery available for use as part of the functionality of the extension, as well. For example, a YACC grammar usually requires the type of the semantic values associated with nonterminals to be pre-declared (as uninterpreted strings of C types.) If it were deemed an improvement, one new possibility is eliminating these declarations, and instead inferring the type for a nonterminal from the type returned by production rules (and checking consistency.) This is hardly something that an external DSL could not do before, it is simply that because we are making direct re-use of the host language, it is now quite easy to accomplish.

As an extension that passes the modular analyses, an external DSL could now be composed with other DSLs re-cast as extensions as well. Using several together opens up possibilities that were not readily available before. In addition, the external DSL re-cast as language extension becomes itself an extensible language. It is now possible for extensions to be written for our parser generator extension (e.g. introducing EBNF.) These extensions are simply subject to the same restrictions, as though YACC were a part of the host language.

Finally, many language extensions are possible that would be considered entirely

impractical as external DSLs. For example, regular expressions *could* be placed in a separate file that is parsed, analyzed, and compiled to DFA at compile time, but doing so would be considered unwieldy. Typically, these are instead embedded as strings within a C program, to be compiled at runtime instead. Implemented as a language extension, it would be possible to deal with these DSLs at compile time, without the overbearing ceremony of writing each small regular expression in a separate file, and preprocessing them with a separate tool.

Thus, we can transform any external DSL into a reliably composable language extension. But, as soon as we begin to change the DSL to take advantage of the new opportunities being a language extension presents, we may then run into issues with the analyses. But at least in some cases, this only affects how we can make the extension work, not whether it can work.

7.2.2 Improving upon embedded DSLs

Embedded DSLs (or "eDSLs") already work within the bounds of an unextended language, and so do not need "conversion" to a language extension. Instead we consider only how realizing the DSL as a language extension offers ways to improve an existing embedded DSL.

The most obvious first benefit is the possibility of using custom syntax, rather than having to find a way to make the existing host language syntax work. This can have significant advantages for both the users of such an eDSL and its implementors. Many eDSLs can be distorted by attempting to fit their design into the constraints necessary to use some host language syntax. For example, some Haskell eDSLs introduce types that claim to be Monad instances but are very much not actually Monads–they simply want to use '`do`' notation.

A second benefit of language extensions over eDSLs is that to create an embedded DSL requires the host language to support many specific language features. Embedded DSLs often take advantage of operator overloading, lambdas, currying or partial application, and the easy ability for expressions to construct data structures (that the eDSL will often later interpret in some fashion.) As a consequence, a language like C is usually considered unsuitable for writing an embedded DSL, instead eDSLs mostly appear for Haskell, Scala, or Ruby. These features are often only necessary because the DSL is trying to use the *runtime* semantics of the host programming language in order to compute its own meaning. A language extension can instead perform these computations in the compiler, unconstrained by the runtime semantics of the language.

Finally, language extensions are capable of introducing novel analysis. Embedded DSLs are only capable of analysis that can be embedded somehow into the host language's type system. These embeddings can be very complex, and result in inscrutable error messages. The error messages templates produce in C++ are a rather notorious example. This constraint may even affect the design choices an embedded language makes. For example, many parser combinator libraries use *ordered* choice between productions, rather than unordered, perhaps in part because they cannot easily detect any ambiguities that would result from the later.

Thus, opting for a language extension over an embedded DSL can alleviate "abuse" of the host language, provide better opportunities for analysis, improve error messages, and permit more suitable syntax.

7.3 Some extensions and their limitations

In the previous section, we have seen that large classes of useful language extensions satisfy the requirements for reliable composition. However, many of the identified approaches to improving upon these classes of extensions begin to push them in directions where the restrictions may come into play. In this section, we use several mini-case studies of language extensions to observe what kinds of extension we can write, what sorts of limitations we encounter in practice, and how these limitations can be in some cases be circumvented. This approach of using example language extensions provides a less precise description of the restrictions than the definitions of the modular analyses, but it does provide a more intuitive and high-level sense of the implications of the modular analysis and non-interference restrictions.

We present several different language extensions, and document the design choices we made for each extension to get them to be composable. For some extensions, we will suggest alternative designs that provide different trade-offs, or designs that were not possible given the plain C host language and the restrictions each extension must live within. We have chosen this set of extensions not to promote them as especially novel or useful, but because each explores different aspects of the limitations necessary for reliable composability.

7.3.1 Conditional tables

Our first extension is a relatively simple one that adds *conditional tables* as found in the RSML^{-e} [98] requirements specification language. It allows writing relatively complex boolean conditions in a form that is sometimes much more readily understandable. Consider the following example:

```
bool implies(bool a, bool b) {
```

```
  return table { a : F T
                 b : * T  };
}
```

The `table` marking terminal provides an unambiguous transition from host language expressions to the extension syntax. This extension recognizes '`T`', '`F`', and '`*`' as terminal symbols distinct from C variables or operators, even though their regular expressions overlap. These are effortlessly disambiguated, thanks to context-aware scanning. After an expression and trailing colon have been consumed, the parser is in a pure-extension LR-parse state where these new terminal symbols are valid, but host language identifiers and operators are not, thus resolving the ambiguity in the scanner. This extension also uses different whitespace rules from C: a newline marks the end of a condition list and the beginning of the next possible C expression. We are constrained syntactically in that the '`:`' token following host-language expressions, while in this case quite natural, must already be in the follow set for expressions (in this case, due to the ternary conditional operator.)

Semantically, each column represents one possible way the whole table expression could be true. The `T`, `F`, and `*` truth flags indicate if the expressions at the beginning of that row must be, respectively, true, false, or either. If all expressions have the appropriate value for the flag, then the column is considered to be true. The table expression evaluates to true if any column does. This example, then, evaluates to true if `a` is false (the first column) or if both `a` and `b` are true (the second column) and otherwise evaluates to false.

This extension demonstrates well the freedom extension developers have within their own custom syntax. An abbreviated version of what the specification for this extension looks like can be found in figure 7.1. On the abstract syntax side of the

Concrete syntax additions

```
marking terminal 'table';
terminal 'T', 'F', '*';
terminal NL /\n/;
ignore terminal WS /[\t\r\ ]+/;

Expr_c ::= 'table' '(' Rows_c ')'
Rows_c ::= Row_c  |  Row_c NL Rows_c
Row_c ::= Expr_c ':' TruthFlag_c
TruthFlag_c ::= 'T' | 'F' | '*'
```

Abstract syntax additions

```
production table
e::Expr ::= t::Rows
{
 forwards to
  if null(t.errors) then
   t.translation
  else
   errorExpr(t.errors);
}
```

Figure 7.1: An abbreviated and simplified excerpt of the Silver code implementing the conditional tables extension.

extension, we see a common pattern for many language extensions. There are no real constraints imposed on how an extension analyzes its own constructs, and so most of this is elided in the figure, though we can see the bridge production makes use of an `errors` and `translation` attributes. If analysis of the new syntax suggests there are no errors (such as mismatching numbers of true/false flags), we compute and forward to a host language tree of the expected nest of C `&&` and `||` operators that is equivalent to this expression. If there are errors, we instead forward to a production in the host language that represents an arbitrary erroneous expression.

One last aspect of this extension's implementation (not shown in the figure) is that it's made reasonable by the use of GCC extensions to C that are a builtin part of the AbleC host language. A naive translation of this extension might result in re-evaluating each expression more than once, a problem if the expressions contain side-effects. This is much easier to handle using GCC "statement-expressions," which allow statements to appear inside expressions. Since a table is an expression, this allows us to forward to an expression that internally declares several local temporaries,

computes each value involved (thus evaluating each expression just once,) and then evaluates the condition table using those temporary variables. Thus, we might emit code like '`return ({int t1 = a; int t2 = b; (t1 && t2) || !t1});`' for our earlier example of a table. Even with this simple example, we can see that a straightforward translation used `t1` twice. Without this GCC extension, we would not have been able to declare these temporaries, since this expression must (of course) forward to an expression. In this case, the extension would still have been possible, but we would need a more complicated implementation that more carefully arranged the checks to ensure each was evaluated only once (and this would potentially entail some code duplication on different branches.)

7.3.2 Regular expressions

Another small domain-specific extension adds Perl-inspired regular expression literals and a corresponding regex matching operator, as in the following small example:

```
if( str =~ /[A-Za-z][A-Za-z0-9_]*/ ) { ...
```

which checks if the string in `str` matches the given regular expression.

This extension is built on top of an existing library (in this case, we chose POSIX '`regex.h`'). This is typical of many language extensions: there will be an existing library, but using it involves clumsy syntax or poor safety properties, for example, and thus motivates using a language extension. In this case, not only does the extension provide more convenient syntax, but it detects errors in the regular expression syntax at compile time, instead of runtime, since the extended AbleC parser will parse the regular expression literal to ensure it is syntactically correct. If we were willing to invest further in this extension, nothing stops us from translating the regular

expression into a DFA at compile time, instead of deferring the process to runtime via the library, however we have not gone that far at this time (and we would need to switch to a different library.)

To implement this extension, we make two new introductions to the concrete syntax. Regular expression literals begin with the marking terminal '`/`', which might be surprising. Despite the C host language's division operator, it turns out the parsing contexts the C grammar generates keep "before an expression" and "before an infix operator" clearly distinct, and consequently there are no lexical ambiguities between the C '`/`' division operator and this extension's '`/`' marking terminal. This marking token is simple enough that it might conflict with other extensions in the future, of course. However, it is still fairly easy to introduce a transparent prefix (essentially, a module scoping construct) to unambiguously reference it. For example, when the user uses two extensions with '`/`', we might instead write '`regex/[A-Z]*/`'.

Like the table extension, regex literals use custom whitespace, making spaces significant and forbidding newlines. The example above also demonstrates another important part of context-aware scanning and custom whitespace: the regular expression ends with '`*/`' which looks like the terminator for a C comment. It's not recognized as such, in part because comments are just another form of whitespace that the regular expression syntax does not allow within the bounds of its forward slashes.

The infix matching operator '`=~`' would not normally be allowed by the modular determinism analysis. It would normally appear *between* two expressions in the right hand side of a production, and the analysis requires extension's marking terminals be the first symbol on the right-hand side of a production. However, we have gotten lucky, and the C standard grammar happens to factor out to a separate nonterminal

the alternatives for comparison operators specifically. The marking terminal '`=~`' is simply an extension to this comparison operator nonterminal, followed by no custom extension syntax at all. The choice of '`=~`' as a marking token is somewhat suspect, however. It would be awkward to have to provide a transparent prefix, for example '`str match=~/[A-Z]*/`'. In general, the syntactic analysis would usually preclude the introduction of new binary operators, unless the host language grammar has taken this unusual step of factoring them out.

For this extension, we have spent much time discussing syntactic restrictions (which, while we can evaluate them on AbleC, are not the primary subject of this thesis.) This is because there are less significant semantic restrictions on this extension. We were forced into a particular design choice, however. AbleC does not offer an easy way to "lift" a declaration to a higher level, nor does C in general offer good ways of automatically initializing things. As a result, this extension represents compiled regexes as static-local variables, initialized on their first use. (For those not familiar with them, local variables declared `static` in C persist between calls of the function, making them essentially global variables but with only local visibility.) But we consider this a perfectly reasonable design choice.

7.3.3 Algebraic datatypes and patterns

The next extension introduces an algebraic datatype declaration, along with new syntax for pattern matching. For example:

```
datatype Type {
 Unit();
 Fun(Type*, Type*);
 Var(char *);
};
```

```
match(t) {
 case Unit() { ... }
 case Fun(l, r) { ... }
 case Var(s) { ... }
}
```

This extension makes use of a marking terminal, '`datatype`', for declaring a new

type along with its set of constructors. A second marking token, '`match`', marks the beginning of a pattern matching statement. The datatype declaration forwards down to a set of declarations that simulate algebraic datatypes using `structs`, `enums`, and `unions` as is done by Hartel and Muller [99] in an extended, though not extensible, version of C that adds algebraic data types. The match statements are (again) able to analyze their own syntax unconstrained by the composition restrictions, in order to ensure that exhaustive cases are present, for example.

One less than perfect aspect of this extension, as we have implemented it, is that it does not prevent access to the internals of a value of an algebraic data type. Although the user is given no indication of what the type's internals are, a user need only look at the generated code to find out. As a result, we do not quite have the guarantee we might want: namely, that the data is only accessed in a safe manner by pattern matching. One possible fix for this problem would be to write a new analysis on the host language that ensures the data is only ever accessed in a safe way. However, our non-interference rules forbid us from recognizing the particular "safe way" of using the pattern matching syntax–a short cut that would make the analysis correctness dependent upon seeing extension syntax prior to forwarding. Instead, the analysis must purely apply to the host language and instead enforce safe access to members of a union by ensuring that all accesses are guarded by a '`switch`' on the union's tag enum, for example. In this way, the '`match`' syntax must be regarded as safe not by dictum, but instead by construction.

A more serious bit of trouble for this extension is that it cannot communicate (directly) the structure of the datatype from declaration to pattern matching statement. Because this information passes from an extension construct in one part of the program, through the host language and its environment, to an extension construct

in another part of the program, non-interference means we cannot rely on seeing anything but host language constructs. Instead, the matching construct must pick apart the host language union-enum-struct structure that a datatype declaration forwards to, in order to discover what cases are possible and their types, for example. This is possible to do, but again it means we must write more involved code that analyzes the host language to reconstruct information we used to already have, instead of simply obtaining it from a higher-level structure.

Finally, the algebraic datatype extension's design is deeply restricted in one serious way: we cannot introduce *parameterized* datatypes, such as `List<a>` or `Pair<a, b>`. The reason for this restriction is the non-interference conditions. C simply does not have an equivalent type we can forward down to. We are able to give semantically equivalent types for non-parameterized types, such as `Type` in our example above, but C does not have any notion of parameterization like templates in C++ or generics in Java.

There is the temptation to evade this problem by using, for example, type erasure and `void *` in the structure the datatype declaration forwards to, but this simply moves the problem elsewhere. The extension would not be able to change the type system's notion of type equality, for example resulting in `List<int>` and `List<double>` being considered equal (they both forward to the same type.) There remains the possibility of C++ style generation of new types, but AbleC has no mechanism to accomplish this, and it cannot be implemented by an extension.

7.3.4 Matrix operations

Another application domain that would benefit from language extension (especially for C) is matrix operations. This is our largest extension, introducing features that

allows the declarative initialization of a matrix, as well as a convenient syntax for operations on a matrix. While it would be nice to re-use C's expression language, there are multiple difficulties in actually doing so when we constraint ourselves to this simple specification of the C host language:

- No operator overloading. Multiplication has a meaning in C and that's that. Plain C (and thus, AbleC) doesn't provide for changing this.
- Limited ability to introduce new operators. Although it worked for the regular expression extension, the C standard grammar doesn't provide for it at the level of arithmetic operators. (It is possible to alter the structure of the grammar without changing the language in order to allow this, but we have not chosen to do so.)
- Memory management difficulties. Operations may involve computing temporary intermediate values, but C lacks some mechanism (like constructors and destructors triggered by scope changes in C++) for cleaning up these temporaries after they are no longer needed [2].
- Non-interference's restrictions on analysis. Associating extra information together with each expression to perform better optimization is difficult when we are faced with a host language type system that has no way to encode it, but trivial for custom syntax.

To get around these issues, the matrix extension introduces its syntax at the level of statements, and includes its own small expression sub-language. The choice

[2]Though, strictly speaking, there do exist GCC extensions for this. And library-based garbage collectors for C also exist. Both are possible workarounds for this particular problem

to use a custom expression sub-language means language extensions that introduce new expressions to C will not be available within the bounds of these special matrix expressions (except perhaps by an escape hatch to C expressions.) A simple example of the extension in action:

```
matrix A_transposed(m, n) =
  { matrix A(n, m) };

matrix let B = A * A_transposed;
```

This declares a new matrix '`A_transposed`' and initializes it to the corresponding values of the matrix '`A`'. Then it declares a new matrix '`B`' to have the value of the product of these previous two matrices.

For an operation like '`matrix C = A * B * v;`', we need to create multiple intermediate temporaries. Because this syntax extension is at the level of a statement, we are able to simply emit the actions that allocate, compute, use, and deallocate the necessary resources as needed.

One somewhat annoying aspect of this extension is the need to use the marking token '`matrix`' when indexing into a matrix (as in '`matrix A(n, m)`'.) We would prefer to be able to do something natural without this extra syntactic noise, but doing so would require C to have some sort of "hookable" feature like operator overloading. Without that, we must resort to a marking token, as we really want to allow indexing into a matrix from arbitrary C expressions.

One major downside to this extension is that we are not able to perform analysis on the number of dimensions each matrix has, and we must allow the possibility of error to persist until runtime. The trouble is that we may provide extra information in a language extension's syntax (such as the dimension or size of a matrix) that we

cannot somehow persist in what it forwards to. As a result, this information cannot be used as an aid in performing a static analysis. We generally refer to this difficulty as the "annotation-driven analysis" problem. We still can try to write an analysis purely of the host language that does this, but doing so is prohibitively difficult, unlike in the previous example where we pursued this approach with pattern matching. Leveraging the custom syntax to discover properties about the code is, in this case, dramatically simpler than purely analyzing the host language and attempting to reconstruct what those properties would be.

7.3.5 Mex functions

As our last example of a language extension, we chose an application area that we think is especially broad and useful. C is the "foreign function" language for many other languages, and so building C code that interfaces with other languages is a common occurrence. Here, we build an extension for MATLAB, but one can imagine similar extensions for other languages such as Python, Lua, Ruby, or Perl. A language extension can automate away much of the relative difficulty of interfacing between languages.

For this extension, we introduce syntax that allows users to declare functions usable within MATLAB via its foreign function interface ('`mexFunction`'). We introduce a single marking terminal '`matlab`' that begins a custom function declaration similar to those in MATLAB. For example:

```
matlab
(unsigned char pic[Y][X][3]) =
  mandelbrot(double xstart, double xend, ...)
{ ... }
```

This extension then produces a function satisfying MATLAB's foreign function interface and extracts the arguments, producing good error messages when the wrong types or number of values are provided. The body of the function is written like any other C function, except for one problem: like the matrix extension, we need special syntax for indexing MATLAB matrixes.

We can resolve this problem in a new way, however. We've been observing all along that our host language could be more extensible if we weren't fixing it as plain C, and it had features like operator overloading. And so, we designed the implementation of the matrix extension's custom syntax to support something like operator overloading. As a result, our mex extension can extend, not just AbleC to add a new kind of function declaration, but also the matrix extension to also understand MATLAB matrix types, in addition to its own. This is an example of extensions extending other extensions.

7.4 Restricted classes of language extensions

We have seen how DSLs can be embedded into the host language with the guarantee that they will safely compose in the way that their independent developers expect. Within these added sub-languages there are no significant restrictions on what the extension can do. They must ultimately yield host language code, but this is essentially no restriction at all for DSLs, as they yield host language code already. It is only in the interface between the sub-language and the C host language where the restrictions come into play, and we have observed some of these difficulties in the example extensions of the previous section. There are two important characteristics of these restrictions:

- They are relative to the host language. Different host languages will have different surrounding "spaces" of potential language extensions.
- They are relative to the host language *implementation*. A C compiler need only recognize C syntax and give it the appropriate semantics. It is possible for the compiler's internal abstract syntax, the truly relevant feature in terms of what extensions are permitted, to be significantly different or more powerful than the host language syntax it accepts. This would enable more extensions.

The Silver implementation of AbleC follows the C standard and Clang implementation where practical, and as such is largely written in the manner that one might used for a traditional non-extensible compiler implementation. That is, there was no effort to foresee future possible language extensions and adapt its design, besides possibly the inclusion of some GCC extensions. As such, we encountered a number of limitations to the kinds of extensions possible for AbleC, which we will now consider more closely. As we will see, almost all of these can be considered limitations of AbleC, rather than our framework for language extension. An AbleC with a modified abstract syntax could permit these extensions. Finally, it is worth keeping in mind that some modifications we propose here are features of languages typically associated with embedded DSLs.

7.4.1 Operator overloading

In our example extensions, the only way to transition from host language syntax to extension syntax is through a marking token indicating an explicit syntactic transition. However, this is not the only possible means of "hooking in" extension constructs. Another approach would be to augment the AbleC abstract syntax (not necessarily

even exposed in the language's concrete syntax) to allow type-based overloading of existing operators.

For example, we could permit matrix operations using the host language expression syntax. When we have an entirely host-language expression such as `a * b`, the host language multiply production can examine the types of the subexpressions, and forward to a handler indicated by those types. (We generally refer to this production as a "dispatch" production.) One such handler would be the ordinary host language multiply production, but extensions types can introduce arbitrary other operations, such as their own forwarding productions. This approach would allow extensions to "hook" into the language based upon the types of expressions, and not just via syntactic marking terminals.

7.4.2 Types and parametric polymorphism

Recall that for the algebraic datatypes extension, we could not permit type parameters to our datatypes. This limitation was caused by our non-interference restrictions, as there was no suitable semantically-equivalent plain C type. One aspect of this worth emphasizing is the existence of multiple "sub-languages" within a typical programming language. While the usual "declaration, statement, expression" phrase structure of the language is less of an obstacle (mostly due to the ability to transition between each of these,) the type language remains essentially separate. Despite C being a very capable programming language in general, its language of types is impoverished: it is not capable of embedding much information at all.

Adding parametric polymorphism to AbleC's type AST is a distinctly probable course for evolving AbleC, as nowadays this is a fairly mature and expected feature

of type systems [3]. Again, this feature could be introduced in a way that does not expose it in the concrete syntax, but makes the feature available for extensions to take advantage of in the abstract syntax. Such an enhancement may be useful for wide variety of extensions and may thus be worth the effort. Enhancing the type system may seem attractive, but it may come at a cost: future changes to the C standard may evolve the language in incompatible directions. Large deviation from the standard could be a liability.

7.4.3 Annotation-driven analysis

Simple examples of the annotation-driven analysis problem can be found with our algebraic datatypes and matrix extensions. For example, we considered what is necessary to implement an analysis that ensures pattern matching expressions have all necessary cases. But we could not do so except by inferring information about the datatype from its union-enum-struct description in the host language.

Perhaps the root of the problem is the inability to express the meaning of an extension's annotation within the semantics of the C host language. That, combined with the need to communicate "across the host language" (instead of purely within extension syntax, where we can perform arbitrary analysis) forces us to communicate only with host language expressible content. In this case, we must communicate from the type declaration to the pattern matching expression, and we cannot describe a datatype except as a "tagged union", and so that is what we must be able to analyze. It is not altogether clear how to solve this problem in a totally generic way, and so this may represent the most accurate criticism of our model of language extension,

[3] Though, how to introduce it *well* into a low-level language like C is probably an open research problem.

rather than just a criticism of AbleC alone. However it is also not entirely clear this problem would not simply be solved by a more sophisticated type system.

7.4.4 Traditional C extension points

Some parts of the annotation-driven analysis problem could be solved by introducing new host language constructs. AbleC already incorporates the GCC extension introducing `__attribute__` annotations. This annotations are an essentially ad-hoc means of introducing compiler extensions. Existing attributes do things ranging from renaming symbols in generated assembly, to indicating a function should be called when a variable goes out of scope, to indicating that a function has no side effects. It's possible that a sufficiently fleshed out implementation of these attributes could solve this problem, by attaching the extra information to ordinary C types.

However, attributes are troublesome in that there is no real specification of their semantics. Currently in practice, compilers essentially handle these attributes via plugins that perform arbitrary mutation on the AST. This is not an approach well-suited to our style of language extension. Taming this mess might result in a system that allows introduction of annotations in such a way that the annotation-driven analysis problem can be solved for language extensions in AbleC.

Presently, AbleC does not give attributes any special behavior at all, they are simply attached to things. They are parsed and pretty printed essentially uninterpreted, for the underlying compiler to worry about. As such, they are not something that is presently useful, except in properly parsing and supporting existing C header files.

To complete this ironic picture of our poor support for traditional extension mechanisms in our extensible compiler, AbleC does not support compiler pragmas. Here,

support is more promising, as compiler pragmas typically come with what is essentially a marking token following the pragma, and so fit our extension model syntactically. However, they vary wildly in terms of semantics. Sometimes they affect the compiler's (or parser's) state, or they change the meaning of the following statement or declaration. As such, we have poor support for these as well, and we only special case a few in order to support existing C headers.

The root of the trouble here is the direct incompatibility of the extension models. Attribute semantics seem based on a plugin able to traverse trees, find annotations, and perform mutation on the abstract syntax tree. We could support such an operation, but we'd be abandoning our assurances about the composability of extensions. (It is worth pointing out that these are generally not extension points for language extensions[4] but rather points where different compilers can accept different variations on the C language. No one appears to be trying to compose these extensions, it is simply that each monolithic compiler may have a different set.) Pragmas seem to likewise simply insert an imperative command to the compiler in the middle of its other operations, and so is a model potentially in conflict with reliable language extension. Thus, it should perhaps not be surprising that these mechanisms do not fit our model, since the lack of reliable composability in existing language extension mechanisms is what motivated our model in the first place.

7.4.5 Code lifting

Many extensions require non-local effects on the abstract syntax tree, and AbleC currently does not provide for such things. Many of these non-local effects, however, fall into common categories.

[4]With OpenMP being a notable exception.

One category is trying to "bubble up" declarations to the top-level from inside even expressions. This was considered as one possible way of trying to twist AbleC into supporting type parameters for algebraic datatypes (essentially the C++ template style.) AbleC does not presently support such a thing, though in some restricted cases, we're also relieved of the need. We support the GCC extensions for nested function declarations, which can sometimes substitute. However this does nothing to help us with, for example, declaring new types that pass out of the scope they're declare within.

Another prominent example of this is executing operations when a variable goes out of scope. For example, in the matrix extension we wished we could make some code run when a variable went out of scope, in order to clean up temporaries. An extension is unable to reach across and manipulate remote parts of the tree, and we do not have any mechanism in to collect code to insert at scope's end. Technically, this need could also be filled by a GCC extension, "`__attribute__((cleanup(fn)))`" which calls `fn` when the variable it is attached to goes out of scope. However, while this is supported by the underlying compiler, it is an attribute uninterpreted by AbleC at this time, and so would not be analyzable by other extensions. (For example, a leak detector static analysis would not necessarily spot the fact that a cleanup call frees a variable.)

7.5 Implications for host language design

In the previous section, we spotted a number of ways in which AbleC and C more generally seemed especially limited for our purposes. We took these limitations as suggestions that could have fed into the language's design.

AbleC already benefited greatly from GCC extensions. Many of these extensions seemed to be designed in order to aid code generators to be implemented more easily and C preprocessor macros to perform feats they might not otherwise be capable of. In this way, it seems that traditional thinking about what makes a language an easy target for code generation is also appropriate for thinking about what makes a language capable of reliable extension.

One of the biggest problems we encountered was the primitive nature of the type system. The lack of parametric polymorphism on the part of C posed a serious issue for fully supporting algebraic datatypes. Historically, both C++ (templates), and Java/C# (generics) have had issues grafting support for polymorphism onto an existing language that lacked it. This lesson in language design earned Go some criticism for not incorporating them from the beginning, as they will almost certainly make for a painful addition in the future.

Our work perhaps suggests a theoretical reason why this sort of change was especially difficult. As we see with AbleC, introducing this change is far outside the bounds of what is possible as a reliable language extension. It is instead a (very difficult) modification to the host language. We conjecture that these sorts of changes to a language are inherently more disruptive.

Leveraging this possibility, we believe that designing a language with extension in mind from the beginning can be hugely beneficial. Already we (as a field) instruct language designers to work "semantics first"[100], discarding consideration of (say) concrete syntax until later. The purposed of this is to focus attention on the pieces that matter most, instead of essentially irrelevant details[5] that many have strong opinions on, but ultimately don't matter. We might further focus our efforts on

[5]Such as the lexical syntax of comments.

substantiative changes to the language, i.e. only those features which cannot be implemented as extensions.

This "modifications first" design approach may have several advantages. First, it further restricts what the language designers pay attention to, leaving what are hopefully more minor issues for later. Second, what is permissible as an extension can inform our language design about what important pieces are missing. If a desired language feature seems ancillary, like it should be implemented as an extension, but the host language at present does not permit it, then perhaps we should turn our attention to what change to the host language would allow it as an extension. Third, it may help guide the development of some ancillary language features. If an extension seems like it should be an extension, and it doesn't *quite* fit, perhaps the design of the extension should be slightly altered to put in more in line with the semantics of the host language. Thus, somewhat quirky designs for language features could be fixed before their slightly odd behavior compounds the complexity of the language. Finally, it tells us as language designers to worry about major features like parametric polymorphism sooner rather than later. Since these features have a history of poorly grafting into the language design later on, doing so seems wise.

This is far from a complete story of how to focus concerns in language design, however. Although our "modifications first" design principle is even more focused than merely starting with the design of the abstract syntax, it still seems far from the typical "calculus style" presentation of a language (as seen with AG in chapter 4.)

7.6 Future work

Further research on AbleC has many possible directions. The most obvious of which is the further development of useful extensions–and so we will say nothing more about that. We will instead take a look at serious obstacles to the actual, practical use of AbleC and reliable composable extensions in practice.

The biggest issue standing in the way is the relatively immature quality of the AbleC front-end compiler. Every piece of software must mature over time, and never appears high-quality from the start. One area for improvement is better supporting under-specified (but nonetheless important) GCC extensions (such as attributes and some pragmas.) Another is the development of a fully accurate C type checker (a somewhat irritating task, as C's type rules do not so much have corner cases as solely consist of corner cases.) Developers also expect higher-quality error messages than it presently provides, and to that end we likely need a full internal implementation of the C preprocessor, with more sophisticated location tracking. AbleC, at present, uses an external preprocessor pass before parsing a file, an approach which is no longer considered good enough.

Next on the list is the overall quality of Silver-generated artifacts (like the AbleC compiler.) This is remarkably good, mostly thanks to fully leveraging the JVM's advantages. However, one area in which we are sadly lacking is the ability to do runtime composition of language extensions. Silver, in fact, already supports the underlying requirements for accomplishing this, and doing runtime composition with Copper has been published[68], however the Copper implementation was for an old version and has atrophied. The overall user interface (how to specify "this language with these extensions") has also not yet been developed.

There are still more pieces to fully realizing language extension in practice. This thesis finishes the first complete story for compilers, but we must still be concerned with other tools. These includes debuggers, documentation generators, static analysis tools, IDEs, and refactoring tools. We can support most code analysis tools as language extensions.

We likely have a good story for debuggers, but doing so would require enhancing AbleC all the way to using LLVM directly to generate machine code–a significant development project. This change would allow concrete syntax to directly introduce debugging information as additional nodes in the abstract syntax they generate. As a result, extensions would be able to indicate how debuggers should handle them, and this would be passed on in the form of debugging information in the compiled binary, to be used by the debugger. This would allow forwarding to continue to do its work as visibly (or invisibly) as the trees they generate say they should.

Silver already has nascent support for generating IDE plugins, but the extension model is not ideal. Generating an IDE plugin requires changes to the host language, because we currently do things like require the terminal declaration to indicate a coloring. Otherwise, it is difficult to ensure that every terminal has coloring information and that some is not lost. (This is similar to the expression problem, except with terminals we don't have a nice solution like forwarding.) Thus, we cannot introduce coloring information separately from the language syntax. As a result we cannot develop an IDE plugin using an external host language without modifying it. It is not clear how best to solve this problem.

Finally, refactoring tools are another important kind of tooling with special problems. These need to directly work with the concrete syntax the user actually wrote, and so we cannot necessarily work via the tree that productions forward to. In the

case of a renaming refactoring, for example, we need to find all instances of that name in the original concrete syntax. Extensions may not have a one-to-one relationship between names that appear in the original syntax and forwarded-to trees. Consider, for example, '`typeof(id)`' (which might forward to simply '`int`') and a request to rename '`id`'. Somehow, this must be spotted, and somehow we must be able to ensure the refactoring will work in the presence of new language extensions.

The last obstacles to the use of AbleC and reliable extension in general involve thinking carefully about problems that may arise as a result of adopting language extension as a technique. One problem users may later face is the desire to stop using a particular language extension, or to transition over to using competitor or new version. It is possible to implement code transformations of these sort as "extensions to the extension," but we have not yet done this sort of thing, nor explored the practical difficulties (e.g. dealing with indentation when rewriting away code.)

Another area of future exploration is the evolution of the host language. We believe there exists a static analysis that would detect whether a change to a host language potentially breaks language extensions, which might help serve as a warning system against unwanted effects, but we have not yet realized such an analysis. Related questions arise about the evolution of a language's design once it hosts many extensions, as changes could not just break extensions but render them no longer possible.

7.7 Related work

There are a variety of tools and techniques for extending programming languages and compilers, and many language implementations created with such approaches. Below

we discuss various categories of these tools, focusing on those works most closely related the work presented here.

Traditional Compilers: These are usually built to allow compiler engineers to add new analyses and optimization passes. Examples of these include the Rose compiler framework [101], the Glasgow Haskell compiler (GHC), Cil [102], and LLVM and its associated C compiler, Clang. While some of these even allow programmers to turn on or off various integrated extensions, they do not support modular extensions that introduce new syntax in any meaningful way beyond directly modifying the compiler. Even then, the recommended way of doing so is often by restricted methods like introducing attributes and pragmas, and not by changing the grammar of the language.

Extensible language frameworks: There are several extensible language tools that similarly use declarative formalisms such as context-free grammars, attribute grammars, or term rewriting rules. While these often allow modular specification of language features, and some even allow composition, these all lack any means of ensuring that composition will succeed provided by our modular analyses. There has been substantial work using attribute grammars for the modular development of languages [103, 104, 105, 106, 79], but this work is primarily aimed at helping compiler engineers re-use language specifications in designing extended languages, and not aimed at programmers looking for tools that would perform reliable composition.

The JastAdd [107] attribute grammar system, in which reference attributes were developed, also supports higher-order attributes [56], and circular attributes. It was used to specify an extensible Java compiler [81] that supports modular language

extensions. However, JastAdd does dynamic, runtime checking for missing equations, and has no static analyses that provide guarantees along the lines of the modular well-definedness analysis in Silver. Its parser generator uses a traditional (non-context-aware) scanner in an LALR(1) parser, and would thus suffer from lexical ambiguities in trying to compose extensions.

The Spoofax [108] language workbench is a similar system but uses Stratego-based term-rewriting and a variety of small domain specific languages for specifying language syntax and semantics. It uses scannerless generalized LR (SGLR) parsing, and thus can generate a parser for any context-free grammar, but this approach provides no guarantees that the composed grammar is not ambiguous. However, by eschewing a traditional scanner and parsing down to the character level, SGLR can handle over-lapping keywords in a manner not unlike a context-aware scanner.

SugarJ [44] is an extensible specification of Java built using Spoofax that aims to provide language extensions as libraries that are simply imported into a program. SGLR parsers are constructed when the extended program is compiled, and are cached to avoid rebuilding on every compilation. While the library-based model with runtime composition is a major advantage of SugarJ, it suffers the same drawbacks mentioned for the Spoofax language workbench. AbleC currently needs to be rebuilt for each change in the accepted extensions. Silver actually does separate compilation and is capable of runtime composition of extensions, and the theoretical problems have been solved for runtime composition of parse tables in Copper [68], but we are missing a good implementation of the later, and so AbleC is not yet able to do this.

AbleC is not the first extensible C compiler. Most notable are XTC [94] and Xoc [95]. XTC supports modular specification of syntactic extensions to C using parsing expression grammars (PEGs). These are similar in form to context-free grammars

but compose using *ordered-choice* for productions with the same left hand side. Johnstone et al. [109] note that "PEGs are a recent reintroduction of Aho and Ullman's TDPL formalism [110, Chapter 6]" and are somewhat difficult to reason about since ordered-choice silently removes all ambiguities. In particular this creates a problem for a programmer composing extensions, as they must choose a correct ordering (and there may not even be a correct ordering) in which to compose extensions.

AbleC shares the same goals of programmer-directed composition of language extensions as Xoc [95], another extensible C compiler. Xoc uses an attribute grammar-like mechanism for semantic analysis and uses GLR-based parser, so extension specifications have a form similar to those in AbleC. But Xoc again provides no assurances that the extensions will reliably compose, in syntax or semantics, and grammar ambiguities are presented to the programmer as a sort of syntax error.

Projectional editing systems: MPS [34] and Intentional Programming [48] are extensible language systems that avoid the need for a parser by using a special editor that lets one alter ASTs directly, even though it looks as if one is editing a text file. While this avoids the challenges of building composable parser specifications, it does lock the programmer into a specific editing tool, whereas AbleC allows programmers to use the text editor of their choice. The MBEDDR [96] tool is an extensible C translator built using MPS that adds a number of extensions for building embedded systems. It does, however, modify the syntax of C, for example `int a[]` is written as `int[] a` instead, whereas AbleC sticks with standard C syntax. A further sticking point for MPS is that it does not offer a solution to the expression problem, instead opting for an object-oriented solution, and so adding new analysis may involve changing the host language implementation.

Additional approaches: Macros systems (traditional, hygienic, etc.) [111] and embedded domain specific languages [7] allow new language constructs to be added but are limited to the existing syntactic forms of the language. Macros systems do not offer a solution to the expression problem, and attempting to actually perform analysis on program trees creates potential composition problems with other macros. For embedded DSLs, new analysis is restricted to what can be embedded into the host language's existing type system, and these embeddings typically result in difficult to understand error messages.

The Delite project uses Scala as a host language for embedding domain-specific constructs and lightweight modular staging [49], a type-based approach to multi-stage programming, to analyze and optimize DSLs embedded in Scala. Delite comes with no ready solution for the expression problem, but the approach is potentially flexible enough to simply adopt by convention one of Scala's solutions for the problem. While this approach avoids an external translator such as AbleC, it does require a sophisticated host language that supports multi-stage programming.

Extended, but not extensible, variants of C: Compiler frameworks such as those mentioned above can be used to design extended versions of C that are not then designed to be further extended. Hartel and Muller [99] have designed an extended version of C that supports algebraic data types, but no further extensions. This work was the motivation for our composable specification of algebraic data types described above. TOM [112] add such data types and associative and commutative pattern matching to multiple target languages, including C. But the extensions are not as well integrated with the host language and are not parsed with the rest of the program. Another example is the original Cilk [91] implementation, an extension to

C with parallel programming constructs.

Safe extensibility: While most extensible language frameworks seems to value expressiveness over reliable composition, not all do. Wyvern [35] is the only extensible language system besides AbleC, to our knowledge, that supports *reliable* composition of independently developed language extensions, at least syntactically, without abandoning parsing in favor of projectional editing. It uses a type-directed, white-space sensitive parsing technique that uses indentation to isolate program fragments to be parsed by a language extension. Or alternatively, Wyvern extensions can also be wrapped in balanced braces, parenthesis or quotations which ensure the same isolation constraint. This is a similar restriction on syntax to the one we describe for embedding external DSL syntax as language extensions. The major difference is that Copper distinguishes extensions syntactically, where Wyvern leverages type information to do so. The extension parser to be used for Wyvern extensions is chosen based on the type of the expressions. This parser runs after the initial parsing step which skips over the language extensions. This ensures extensions' syntax can be composed safely, but the approach does not accommodate introducing new analysis of the host language.

With the exception of Wyvern, the various systems described above provide expressive means for defining extensible languages but they do not provide the strong guarantees of reliable composition of language extensions that is our primary focus with AbleC.

7.8 Conclusion

AbleC is, to our knowledge, the first extensible compiler for a mainstream language that supports *reliable* composition of independently-developed language extensions that add both new syntax and new semantics to the host language. This assurance applies to not just concrete syntax and the generation of a scanner and parser, but also to the semantic analysis and translation phases. Of course, this guarantee comes with restrictions on what can be added as a composable language extensions, but we have demonstrated that interesting language extensions are still possible under these restrictions.

It is our working hypothesis that the lack of reliable composition has been an impediment for both users and developers of language extensions, and that this work enables both. Without this assurance, users could not be certain language extensions would work without investing time and effort into using them only to discover a fatal conflict afterward. Similarly, extension developers could not hope to be able to ensure the correctness of their extensions, when each new extension in the ecosystem grows the space of interactions exponentially. Finally, having a clear dividing line between potential extensions versus those that constitute changes to the host language helps even host language designers. They can now reason about the extensibility of their language, and focus their efforts on those language features that could not be provided by an extension.

AbleC provides a good platform for experimenting with new language features, and we are finding it useful for exploring abstractions for parallel programming in particular. In this problem domain, there seem to be many good domain-specific solutions but few general purpose ones. An ecosystem of language extensions solving

different types of parallel programming problems frees us from the need to search for a possibly non-existent silver[6] bullet. An extensible platform like AbleC lets language researchers easily design new abstractions for parallel programming but, perhaps more importantly, easily allows programmers to experiment with them in realistic settings and thus provide better feedback. New parallel programming features that are locked up in an entirely new language may be less likely to find as many users willing to evaluate them in practice.

Users can more easily try new extensions because AbleC provides a smooth evolutionary path for an application written in plain C to move onto C extended with new domain-specific language features. Users can switch compilers to AbleC with their existing program as-is, pull in some potentially useful extensions, and evaluate the use of the new language features with no large up-front costs. This sort of evolutionary approach isn't entirely novel, but AbleC makes it easy for extension developers: every developer does not need their own C-compatible programming language. The ability to compose extensions also makes the platform more attractive to users, as there can now be several features they would gain by making a single transition of tools, rather than several incompatible options each with different advantages and disadvantages.

Many of the limitations we encountered were caused by restrictions relative to the host language, and the fact that plain C (even with some GCC extensions) was a somewhat impoverished host language, extensibility-wise. The modular analyses and non-interference principles provide the first natural, useful and clear theoretical dividing line between language features that are potential language extensions versus those that constitute a (non-composable) change or modification to the host language.

[6] **cough**

We believe this could play an important role in the future when it comes to both the practice and theory of the design of programming languages. It also opens up new questions about the design of languages with (reliably composable) extensibility in mind, on which we have as yet only scratched the surface.

AbleC is available at `http://melt.cs.umn.edu/ableC/`.

Chapter 8

Conclusion

Our original working hypothesis is that the inability to safely and reliably compose language extensions was one of the key problems standing the way of practical use of extensible languages. Other approaches we surveyed in chapter 2 that permitted safe composition were often seriously limited: lacking the ability to introduce either new analysis or syntax, producing extremely poor error messages, limited to a single host language, or suffering significant runtime overhead. This thesis provides the last major pieces necessary to reliably compose languages extensions into a working compiler, without suffering these serious limitations.

However, we still must live with some (less serious) limitations. First, while this is a candidate solution for *compilers*, we have not yet solved the problem for other kinds of language tooling: debuggers, profilers, or refactoring tools, for example. This does stand in the way of practical adoption of this technique, until compatible solutions are found for those tools as well. Second, the modular analyses we impose on extensions also constitute limitations, although this only restricts the kinds of extensions that can be built using these tools. And at least some of these observed restrictions are at least partly due to attempting to extend an existing (and relatively primitive) host language, which was not designed with extension in mind. A better host language would be much more amenable to extension. Moreover, some limitations of this form are necessary: it is not possible to ensure the successful composition of just any

language extension. So these limitations are also a feature: given a host language, they define a precise space of extensions that can be built for this language, all of which compose together flawlessly.

This thesis contributed two major pieces of the solution to reliably composing language extensions. First, we developed the *modular well-definedness analysis* on attribute grammars with forwarding. This analysis ensures we are able to successfully compose any number of passing extensions into a well-defined attribute grammar. Second, we developed a modular approach to enforcing *non-interference* between language extensions that we call *coherence*. Coherence gives us the tools we need to specify the behavior of a language extension in a modular way, and ensure these specifications (and their proofs) will still apply to the composed compiler as well. These contributions, together with prior work providing a modular determinism analysis for composing context-free grammars, provide a candidate solution to the problem of reliable composing language extensions into a host compiler.

The key to all of these contributions is a *modular analysis*. We show global properties (about the composed language) by ensuring only local properties hold of each component (such as each language extension). This is typically accomplished by identifying and preventing extensions from modifying certain key facts, and then exploiting the invariance of these facts to prove the desired global property. This allows us to reduce blunt restrictions (such as not permitting new syntax) into more precise and weakened restrictions (such as not permitting new *non-forwarding* productions). Thus, we significantly improve the potential space of extensions, and we are able to do so without requiring a closed-world of extensions.

8.1 Central contribution: modular well-definedness

In order to ensure that composition of language extensions succeeds, it is necessary to ensure certain classes of errors cannot be created by the composition process itself. Chief among these is the *expression* (or *independent extensibility*) *problem*: independent extensions each introducing syntax and analysis create potential areas of conflict. Without a solution to this problem, composition will fail without glue code to resolve the conflict. Attribute grammar forwarding is helpful here, but the mere presence of this language feature is not enough to ensure the success of composition.

The modular well-definedness analysis (chapter 5) solves this problem by requiring certain properties hold true of the module import/export graph. This captures in a more detailed way the essential difference between a "host language" and its extensions. We fix two pieces of information: the set of non-forwarding productions for each nonterminal, and the flow type of each synthesized attribute occurrence (which captures its potential dependencies). These values cannot be changed by extensions, and as a result, we are able to ensure attribute grammar completeness in a modular way.

Moreover, we are able to infer these values, rather than require them be explicitly provided. Although this is fairly obvious for the set of non-forwarding productions, we show that inferring the flow types is a modest modification of the traditional attribute grammar flow analysis (the principle new addition involving treated some edges as "suspect"). This minimizes the burden on the language extension developer.

As a result, we are able to take a host language and any number of extensions that pass the modular well-definedness analysis, automatically compose them, and we know that the composed attribute grammar will be complete.

8.2 Central contribution: non-interference through coherence

Beyond the composition process itself, we must also solve the problem of reasoning about the behavior of the resulting composed compiler. Not only can composing language extensions together invalidate properties we wish would hold for the composed compiler, it can leave gaps in the specifications with no guidance on what the intended behavior should be. This is actually the expression problem all over again, just lifted up to the level of logical specification, rather than implementation of the attribute grammar.

Coherence resolves this problem in a similar fashion to forwarding: we must ensure our specifications for forwarding productions are semantically equivalent to the tree they forward to. This makes it reasonable to automatically fill in the gaps in our specifications that arise from composition. Even better, it is enough to ensure that the verification of that specification still holds for the composed compiler. This allows us to modularly verify language extensions, which means we've eliminated the possibility of interference between extensions.

This approach is flexible enough to permit custom error messages, arbitrary subtree transformations, and custom pretty-printing for new syntax. Remarkably, though coherence concerns the verification of an attribute grammar, we are able to give a (QuickCheck-based) testing procedure that is highly likely to discover incoherence, without any specification at all. As a result, we are able to run automatic tests on language extensions, and discover possible ways which an extension might interfere with another, without any knowledge of any other extensions, and without requiring extension developers to actually perform formal verification.

8.3 Other contributions

As part of achieving the above two central contributions, we make a number of other contributions as well. The development and integration of polymorphic type system for attribute grammars is not entirely novel on its own, but our design distinguishing and deliberately exposing decorated and undecorated types (as well as their use in code disambiguation) is apparently novel. We further contribute a method of pattern matching on attribute grammars that does not compromise the extensibility of the object language by developing a semantics for pattern matching that is entirely consistent with attribute access (and forwarding in particular.) This work is all presented in chapter 4.

We have also developed AbleC, a (reliably) extensible C compiler, presented in chapter 7. We hope this will serve as a good platform for further research. We built several extensions for AbleC, in part as evaluation of our set of restrictions on language extensions for a real-world host language. Finally, we showed that large classes of DSLs can be re-cast as language extensions which pass our modular analyses, and we note some obvious ways in which they can typically be improved.

8.4 The road ahead

Our work on AbleC represents a synthesis of the major contributions of this thesis. As a result, its future work discussion (in section 7.6) is also a good one for this thesis as a whole. From that section, there are a few broadly interesting problems that merit bringing up again.

First, though we have developed the theory and a practical C compiler that supports reliable composition of language extensions, we note that there remain chal-

lenges for other tools beyond just the compiler. These include debuggers, IDEs, and refactoring tools especially. We suspect the problems posed by language extension are solvable for debuggers, however IDEs and refactoring tools are presently without a good path to a solution. Debuggers are an easier problem because it's reasonable for a host language to have to address them. Refactoring and IDE support are things that are harder because (1) they seem like things that should be extensions, not part of the host, and (2) concern themselves with the concrete syntax as-written, not with post-transformation abstract syntax trees. It is harder to figure out how this should be handled.

Second, we have also left largely unexplored the area of developing a host language with reliably composable language extension in mind. As we have noted, all of our restrictions imposed by our modular analyses (including Copper's analysis on syntax) are relative to the host language. As a result, host language design can have a large impact on the space of possible extensions (see discussion in section 7.4). As a practical direction, this work might include attempting to adapt an existing host language (like C) to better accommodate language extensions.

Besides AbleC, we should also refer back to the future work mentioned for our approach to non-interference (section 6.9.1). Notably, coherence is just one means to the non-interfering end. It is possible there are other approaches out there.

Both coherence and the modular well-definedness analysis have some warts left over that could merit solving. For interference, there is the problem of generalizing the notion of coherence from mere properties to logical propositions in general. If the goal is to do actual verification, instead of just attribute property testing, then this is probably a prerequisite. For modular well-definedness, there is the problem of the overly-conservative solution to dealing with reference attributes. This conservative

solution does not ever allow extensions to pass new inherited attributes through references.

One concluding remark we'd like to make about chapter 4: attribute grammars remain an interesting stylistic hybrid between object-oriented and functional programming. Although we've used Ag as a tool for exploring language extension in particular, we think there is a lot of room for exploring language design in general by experimenting on Ag.

Finally, another remaining direction is to fully exploit the potential major application areas for language extension. We have so far largely produced models of extensions, not the kinds of truly interesting applications that we hope to get from language extension. For AbleC especially, we believe a promising direction is to build extensions for very different kinds of parallel programming models. Being able to integrate a variety of such models into one language could provide a useful playground for experimentation. (GHC has done something like this for Haskell, but it is a closed-world, consisting of only three primary "extensions" integrated into the compiler). A second area where language extensions seems especially good for AbleC is in the development of extensions supporting the foreign function interfaces of various programming languages. We have built a prototype one for Matlab, but C is the foreign function language of most programming languages and each could be well served by a language extension. Finding more areas where language extension opens up new design possibilities could be an exciting area for future research.

Bibliography

[1] S. Peyton Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98. Available at URL: `http://www.haskell.org`, February 1999.

[2] Martin Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

[3] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the .net framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, New York, NY, USA, 2006. ACM Press.

[4] Jeffery Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. Symp. on Operating System Design and Implementation (OSDI)*, 2004.

[5] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[6] S.C Johnson. Yacc - yet another compiler compiler. Technical Report 32, Bell Laboratories, July 1975.

[7] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), December 1996.

[8] S.D. Swierstra. Combinator parsers: From toys to tools. *Electronic Notes in Theoretical Computer Science*, 41(1):38 – 59, 2001. 2000 ACM SIGPLAN Haskell Workshop (Satellite Event of PLI 2000).

[9] Anthony M. Sloane. Lightweight language processing in kiama. In *Proc. of the 3rd summer school on Generative and transformational techniques in software engineering III (GTTSE 09)*, pages 408–425. Springer, 2011.

[10] Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute grammars fly first-class: How to do aspect oriented programming in haskell. In *Proc. of the Int'l Conf. on Functional Programming*, 2009.

[11] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, pages 7:1–7:8, New York, NY, USA, 2012. ACM.

[12] Philip Wadler. The expression problem. Email, November 1998. Discussion on the Java Genericity mailing list.

[13] Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *In Proc. FOOL 12*, 2005.

[14] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, , and Martin Odersky. Optimizing data structures in high-level programs. In *POPL*, 2013.

[15] Arch D. Robison. Impact of economics on compiler optimization. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, JGI '01, pages 1–10, New York, NY, USA, 2001. ACM.

[16] P. W. Trinder, K. Hammond, J. S. Mattson, Jr., A. S. Partridge, and S. L. Peyton Jones. Gum: a portable parallel implementation of haskell. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, PLDI '96, pages 79–88, New York, NY, USA, 1996. ACM.

[17] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 295–308, New York, NY, USA, 1996. ACM.

[18] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP '07, pages 10–18, New York, NY, USA, 2007. ACM.

[19] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, August 2008.

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

[21] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th Intl. Conf. on Compiler Construction*, volume 2304 of *LNCS*, pages 128–142, 2002.

[22] August Schwerdfeger. *Context-Aware Scanning and Determinism-Preserving Grammar Composition, in Theory and Practice*. PhD thesis, University of Minnesota, Department of Computer Science and Engineering, Minneapolis, Minnesota, USA, 2010.

[23] D. V. Schorre. META II : a syntax-oriented compiler writing language. In *Proceedings of the 1964 19th ACM national conference*, ACM '64, pages 41.301–41.3011, New York, NY, USA, 1964. ACM.

[24] R. S. Scowen. Babel, an application of extensible compilers. *SIGPLAN Not.*, 6(12):1–7, September 1971.

[25] Jean D. Ichbiah. Extensibility in simula 67. *SIGPLAN Not.*, 6(12):84–86, September 1971.

[26] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in clu. *Commun. ACM*, 20(8):564–576, August 1977.

[27] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An experimental applicative language. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, LFP '80, pages 136–143, New York, NY, USA, 1980. ACM.

[28] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schuman, editor, *New Directions in Algorithmic Languages 1975*, pages 157–168, Rocquencourt, France, 1975. IFIP Working Group 2.1 on Algol, INRIA.

[29] William R. Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 557–572, New York, NY, USA, 2009. ACM.

[30] Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses - practical extensibility with object algebras. In *ECOOP*, pages 2–27, 2012.

[31] Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008.

[32] Benjamin Delaware, Bruno C. d. S. Oliveir, and Tom Schrijvers. Meta-theory à la carte. In *POPL*, 2013.

[33] Mps user's guide. Available `http://confluence.jetbrains.net/display/MPSD25/MPS+User%27s+Guide`.

[34] M. Voelter. Language and ide modularization, extension and composition with mps. In *GTTSE 2011*, 2011.

[35] Cyrus Omar, Benjamin Chung, Darya Kurilova, Alex Potanin, and Jonathan Aldrich. Type-Directed, Whitespace-Delimited Parsing for Embedded DSLs. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, GlobalDSL '13, pages 8–11, New York, NY, USA, 2013. ACM.

[36] Alessandro Warth. *Experimenting with Programming Languages*. PhD thesis, University of California - Los Angeles, 2008.

[37] JACQUES CARETTE, OLEG KISELYOV, and CHUNG-CHIEH SHAN. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19:509–543, 2009.

[38] Tijs van der Storm, WilliamR. Cook, and Alex Loh. Object grammars. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering*, volume 7745 of *Lecture Notes in Computer Science*, pages 4–23. Springer Berlin Heidelberg, 2013.

[39] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 Object–Oriented Programming*, volume 1241 of *LNCS*, pages 220–242, 1997.

[40] G. Hedin and E. Magnusson. JastAdd - an aspect oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[41] Arthur Baars, Doaitse Swierstra, and Andres Loh. Uu ag system reference manual. http://www.cs.uu.nl/ arthurb/data/AG/AGman.html, 2004.

[42] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping, An Algebraic Specification Approach*. World Scientific, 1996.

[43] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the asf+sdf compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, July 2002.

[44] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: library-based syntactic language extensibility. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 391–406, New York, NY, USA, 2011. ACM.

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

[46] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: a domain specific language for source code analysis and manipulation. In *Proc. of Source Code Analysis and Manipulation (SCAM 2009)*, 2009.

[47] C. Simonyi. The future is intentional. *IEEE Computer*, 32(5):56–57, May 1999.

[48] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. *SIGPLAN Not.*, 41(10):451–464, 2006.

[49] Tiark Romph and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proc. of the ACM SIGPLAN 2010 Conf. on Generative Programming and Component Engineering (GPCE '10)*, pages 127–136, New York, NY, USA, 2010. ACM.

[50] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

[51] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161. ACM Press, 1986.

[52] M. V. HERMENEGILDO, F. BUENO, M. CARRO, P. LÓPEZ-GARCÍA, E. MERA, J. F. MORALES, and G. PUEBLA. An overview of ciao and its design philosophy. *Theory and Practice of Logic Programming*, 12:219–252, 1 2012.

[53] Daniel Cabeza and Manuel Hermenegildo. A new module system for prolog. In *Computational Logic—CL 2000*, pages 131–148. Springer, 2000.

[54] Uwe Kastens. Ordered attributed grammars. *Acta Informatica*, 13:229–256, 1980. 10.1007/BF00288644.

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

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

[57] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239—250, January 1984.

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

[59] John Tang Boyland. Remote attribute grammars. *J. ACM*, 52(4):627–687, 2005.

[60] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. *ACM SIGPLAN Notices*, 21(7), 1986.

[61] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39–54, January 2010.

[62] Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. Attribute grammar-based language extensions for Java. In *European Conf. on Object Oriented Prog. (ECOOP)*, volume 4609 of *LNCS*, pages 575–599. Springer, 2007.

[63] E. Van Wyk and E. Johnson. Composable language extensions for computational geometry: a case study. In *Proc. 40th Hawaii Intl' Conf. on System Sciences*, 2007.

[64] Eric Van Wyk and Yogesh Mali. Adding dimension analysis to java as a composable language extension. In *Post Proc. of Generative and Transformational Techniques in Software Engineering (GTTSE)*, number 5235 in LNCS, pages 442–456. Springer, 2008.

[65] Lijesh Krishnan. *Extensible Languages via Modular Declarative Specifications*. PhD thesis, University of Minnesota, Department of Computer Science and Engineering, Minneapolis, Minnesota, USA, 2012.

[66] Eric R. Van Wyk and August C. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *Proceedings of the 6th international conference on Generative programming and component engineering*, GPCE '07, pages 63–72, New York, NY, USA, 2007. ACM.

[67] August C. Schwerdfeger and Eric R. Van Wyk. Verifiable composition of deterministic grammars. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 199–210, New York, NY, USA, 2009. ACM.

[68] August Schwerdfeger and Eric Van Wyk. Verifiable parse table composition for deterministic parsing. In *2nd International Conference on Software Language Engineering*, volume 5969 of *LNCS*, pages 184–203. Springer, 2010.

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

[70] H. Ganzinger and R. Giegerich. Attribute coupled grammars. *SIGPLAN Notices*, 19:157–170, 1984.

[71] Joao Saraiva. *Purely Functional Implementations of Attribute Grammars*. PhD thesis, University of Utrecht, 1999.

[72] K. Backhouse. A functional semantics of attribute grammars. In *7th Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'01*, volume 2280 of *Lecture Notes in Computer Science*, pages 142–157. Springer-Verlag, 2002.

[73] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proc. of the Int'l Conf. on Functional Programming*, pages 50–61. ACM, 2006.

[74] Jimin Gao. *An Extensible Modeling Language Framework via Attribute Grammars*. PhD thesis, University of Minnesota, Department of Computer Science and Engineering, Minneapolis, Minnesota, USA, 2007.

[75] T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *Proc. of the Int'l Conf. on Functional Programming*, pages 341–352. ACM, 2009.

[76] P. Wadler. Efficient compilation of pattern matching. In *The Implementation of Functional Programming Languages*, pages 78–103. Prentice-Hall, 1987.

[77] Ted Kaminski and Eric Van Wyk. Modular well-definedness analysis for attribute grammars. In *Proceeding of the International Conference on Software Language Engineering*, volume 7745 of *LNCS*, pages 352–371. Springer, 2012.

[78] Bernard Lorho and C Pair. Algorithms for checking consistency of attribute grammars. *Proving and improving programs*, pages 29–54, 1975.

[79] Joao Saraiva and Doaitse Swierstra. Generic Attribute Grammars. In *2nd Workshop on Attribute Grammars and their Applications*, pages 185–204, 1999.

[80] Ted Kaminski and Eric Van Wyk. Evaluation of modular completeness analysis on silver. Technical report, Department of Computer Science and Engineering, 2012. Available at `http://melt.cs.umn.edu/pubs/kaminski12tr`.

[81] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[82] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proc. of ICFP*, 2000.

[83] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831—847, 1998.

[84] C. Kästner, S. Apel, and K. Ostermann. The road to feature modularity? In *Proceedings of the 3rd International Workshop on Feature-Oriented Software Development (FOSD)*, FOSD '11, pages 5:1–5:8, New York, NY, USA, 2011. ACM.

[85] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(4):49–84, 2009.

[86] Ralf Lämmel. Grammar testing. In *Proc. of Fundamental Approaches to Software Engineering (FASE) 2001*, volume 2029 of *LNCS*, pages 201–216. Springer-Verlag, 2001.

[87] Ralf Lämmel and Jörg Harm. Testing attribute grammars. In *3rd Workshop on Attribute Grammars and their Applications (WAGA 2000)*, pages 79–98, July 2000.

[88] Arie Middelkoop, Atze Dijkstra, and S.Doaitse Swierstra. Dependently typed attribute grammars. In Jurriaan Hage and MarcoT. Morazán, editors, *Implementation and Application of Functional Languages*, volume 6647 of *Lecture Notes in Computer Science*, pages 105–120. Springer Berlin Heidelberg, 2011.

[89] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2. ACM, 2009.

[90] Florian Lorenzen and Sebastian Erdweg. Sound type-dependent syntactic language extension. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 204–216, New York, NY, USA, 2016. ACM.

[91] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of Programming language design and implementation*, PLDI '98, pages 212–223. ACM, 1998.

[92] NVIDIA Corporation. Nvidia cuda programming guide, 2007. Version 1.1.

[93] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[94] Robert Grimm. Better extensibility through modular syntax. In *Proc. of Conf. on Programming Language Design and Implementation (PLDI)*, pages 38–51. ACM Press, 2006.

[95] Russ Cox, Tom Bergany, Austin T. Clements, Frans Kaashoek, and Eddie Kohlery. Xoc, an extension-oriented compiler for systems programming. In *Proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[96] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. Mbeddr: An extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 121–140, New York, NY, USA, 2012. ACM.

[97] Eric Brewer, Jeremy Condit, Bill McCloskey, and Feng Zhou. Thirty years is long enough: getting beyond c. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 14–19, Berkeley, CA, USA, 2005. USENIX Association.

[98] Jeffrey M. Thompson and Mats P.E. Heimdahl. An integrated development environment prototyping safety critical systems. In *Tenth IEEE International Workshop on Rapid System Prototyping (RSP) 99*, pages 172–177, June 1999.

[99] Pieter H. Hartel and Henk L. Muller. Simple algebraic data types for c. *Software: Practice and Experience*, 42(2):191–210, 2012.

[100] Martin Erwig and Eric Walkingshaw. Semantics first! In Anthony Sloane and Uwe Aßmann, editors, *Software Language Engineering*, volume 6940 of *Lecture Notes in Computer Science*, pages 243–262. Springer Berlin Heidelberg, 2012.

[101] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000, http://www.worldscientific.com/doi/pdf/10.1142/S0129626400000214.

[102] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proc. 11th International Conf. on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer-Verlag, 2002.

[103] H. Ganzinger. Increasing modularity and language-independency in automatically generated compilers. *Science of Computer Programing*, 3(3):223–278, 1983.

[104] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.

[105] R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars. In *19th ACM Symp. on Prin. of Programming Languages (POPL)*, pages 223–234, 1992.

[106] S. R. Adams. *Modular Grammars for Programming Language Prototyping*. PhD thesis, University of Southampton, Department of Elec. and Comp. Sci., UK, 1993.

[107] Torbjörn Ekman and Görel Hedin. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69:14–26, December 2007.

[108] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In *OOPSLA*. ACM, 2010.

[109] Adrian Johnstone, Elizabeth Scott, and Mark van den Brand. Ldt: a language definition technique. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*, LDTA '11, pages 9:1–9:8, New York, NY, USA, 2011. ACM.

[110] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling, volume 1 — Parsing of Series in Automatic Computation.* Prentice-Hall, 1972.

[111] D. Weise and R. Crew. Programmable syntax macros. *ACM SIGPLAN Notices*, 28(6), 1993.

[112] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In *Proc. 12th International Conf. on Compiler Construction*, volume 2622 of *LNCS*, pages 61–76. Springer, 2003.

Appendix A

Formal reasoning about attribute grammars

Some of the discussion of examples of properties in chapter 6 can be confusing for readers who are attempting to understand it concretely, rather than accepting a high-level discussion. In this appendix, we present concrete proofs using Coq of several of the examples used in this chapter. These proof scripts were written for Coq 8.3. Later version may require adjustments to imports and inferred names used in proof scripts. We have attempted to make clear (in comments) what should be happening on any line involving a generated name.

A.1 RepMin of figure 6.3

We do not present a formal procedure for embedding code written AG into Coq, but we believe the following translation of the attribute grammar for RepMin is fairly straightforward. We begin with some import requirements for later.

```
(* We require transitivity of <= *)
Require Import Coq.Arith.Le.
(* 'min' function, and basic facts like commutativity, and cases. *)
Require Import Coq.Arith.Min.

(* Direct translation of RepMin undecorated tree type *)
Inductive RepMin : Type :=
| leaf : nat -> RepMin
| inner : RepMin -> RepMin -> RepMin
```

```
.

(* No forwarding productions *)
Inductive RepMin_fwds : RepMin -> RepMin -> Prop :=
.

(* Evaluation relation of synthesized attribute 'min' *)
Inductive min_on_RepMin : RepMin -> nat -> Prop :=
| min_on_leaf : forall a, min_on_RepMin (leaf a) a
| min_on_inner : forall x y a b,
                 min_on_RepMin x a ->
                 min_on_RepMin y b ->
                 min_on_RepMin (inner x y) (min a b)
.
```

Note that we have a type for representing an undecorated tree, and the evaluation relations for both forwarding and the synthesized attribute `min`. The only interesting part of this translation are the evaluation relations. The flow types for `min` and forwarding both require no inherited attributes, and so these relations directly relate a tree to a result with no other arguments. We will see an example of an attribute that does depend on an inherited attribute later.

Next, we directly translate the custom inductive definitions given by this attribute grammar's verification.

```
(* Definition of inductive relations over trees *)
Inductive lte_all : RepMin -> nat -> Prop :=
| lte_leaf : forall x m, m <= x -> lte_all (leaf x) m
| lte_inner : forall x y m,
              lte_all x m ->
              lte_all y m ->
              lte_all (inner x y) m
.

Inductive exis : RepMin -> nat -> Prop :=
| exis_leaf : forall m, exis (leaf m) m
```

```
| exis_inner : forall x y m,
               exis x m \/ exis y m ->
               exis (inner x y) m
.
```

These are, again, fairly straightforward translations of the syntax from the original figure. Note we have named our relation `exis` here instead of the original name.

We're almost done with this figure, but before we write the proof of the final property, we actually need to show a lemma that we'd glossed over. This lemma is about `lte_all` by itself, essentially just showing that `lte_all` happily accepts any value less than the minimum.

```
Lemma lte_is_min :
  forall t m n,
  lte_all t m ->
  lte_all t (min m n).
Proof.
  intros t m n H.
  (* By induction on the claim that 'lte_all t m' *)
  induction H; intuition.
    (* Leaf case: straightforward, but show property of 'min' *)
    constructor.
    (* show m <= x implies min m n <= x for any n! *)
    apply min_case_strong; intuition. apply (le_trans n m x H0 H).
    (* Inner case: just apply IH *)
    constructor; assumption.
Qed.
```

Finally, we have the proof of our property. In chapter 6 we talked a lot about induction on *decorated* trees, but we don't actually have a type representing decorated trees available! Instead, induction on decorated trees corresponds to mutual induction on the undecorated tree types, and all evaluation relations for it. To prove this property, we will proceed by induction on the evaluation relation of for `min`.

```
Theorem min_correctness :
  forall t m,
  min_on_RepMin t m ->
  lte_all t m /\ exis t m.
Proof.
  intros t m SYN.
  induction SYN.
  (* Leaf case *)
    constructor.
      (* Showing lte_all *)
      constructor. constructor.
      (* Showing exis *)
      constructor.

  (* Inner case *)
  constructor.
    (* Showing lte_all *)
    constructor; intuition.
      (* Take advantage of our lemma about 'lte' and 'min *)
      apply (lte_is_min x a b); assumption.
      rewrite -> min_comm.
      apply (lte_is_min y b a); assumption.
    (* Showing exis *)
      constructor; intuition.
      apply min_case.
        left. assumption.
        right. assumption.
Qed.
```

The proof works in part because of the induction hypothesis we get by induction on the evaluation relation instead of the tree. For the inner case, we end up with the following proof goal:

```
SYN1 : min_on_RepMin x a
SYN2 : min_on_RepMin y b
IHSYN1 : lte_all x a /\ exis x a
IHSYN2 : lte_all y b /\ exis y b
______________________________________(1/1)
lte_all (inner x y) (min a b) /\ exis (inner x y) (min a b)
```

The induction hypotheses come from the definition of `min_on_RepMin`. We're able to show this case because of the definition of these relations on `inner` and these induction hypotheses.

A.2 Extended RepMin of figure 6.4

We begin by modifying the above definitions to introduce additional cases for the new `three` forwarding production. The following is presented as a "diff" against what appeared previously.

```
 (* Direct translation of RepMin undecorated tree type *)
 Inductive RepMin : Type :=
 | leaf : nat -> RepMin
 | inner : RepMin -> RepMin -> RepMin
+| three : RepMin -> RepMin -> RepMin -> RepMin
 .

 (* No forwarding productions *)
 Inductive RepMin_fwds : RepMin -> RepMin -> Prop :=
+|three_fwds: forall x y z, RepMin_fwds (three x y z) (inner (inner x y) z)
 .

 (* Evaluation relation of synthesized attribute 'min' *)
 Inductive min_on_RepMin : RepMin -> nat -> Prop :=
 | min_on_leaf : forall a, min_on_RepMin (leaf a) a
 | min_on_inner : forall x y a b,
                  min_on_RepMin x a ->
                  min_on_RepMin y b ->
                  min_on_RepMin (inner x y) (min a b)
+| min_on_three : forall x y z a b c,
+                 min_on_RepMin x a ->
+                 min_on_RepMin y b ->
+                 min_on_RepMin z c ->
+                 min_on_RepMin (three x y z) (min (min a b) c)
 .
```

```
 (* Definition of inductive relations over trees *)
 Inductive lte_all : RepMin -> nat -> Prop :=
 | lte_leaf : forall x m, m <= x -> lte_all (leaf x) m
 | lte_inner : forall x y m,
               lte_all x m ->
               lte_all y m ->
               lte_all (inner x y) m
+| lte_three : forall x y z m,
+              lte_all x m ->
+              lte_all y m ->
+              lte_all z m ->
+              lte_all (three x y z) m
 .

 Inductive exis : RepMin -> nat -> Prop :=
 | exis_leaf : forall m, exis (leaf m) m
 | exis_inner : forall x y m,
                exis x m \/ exis y m ->
                exis (inner x y) m
+| exis_three : forall x y z m,
+               exis x m \/ exis y m \/ exis z m ->
+               exis (three x y z) m
 .
```

Next, we amend the proofs from before to have a case for the new production.

```
Lemma lte_is_min :
  forall t m n,
  lte_all t m ->
  lte_all t (min m n).
Proof.
  [[ SNIPPED. As before. ]]
    (* Three case: IH again *)
    constructor; assumption.
Qed.

Theorem min_correctness :
  forall t m,
  min_on_RepMin t m ->
```

```
  lte_all t m /\ exis t m.
Proof.
  [[ SNIPPED. As before. ]]
  (* Three case *)
  constructor.
    (* Showing lte_all *)
    constructor; intuition.
      (* x to a *)
      apply lte_is_min; apply lte_is_min; assumption.
      (* y to b *)
    apply lte_is_min; rewrite -> min_comm; apply lte_is_min; assumption.
      (* z to c *)
      rewrite -> min_comm. apply lte_is_min. assumption.
    (* Showing exis *)
      constructor; intuition.
      (* Just discharge all the cases *)
      repeat (try apply min_case; intuition).
Qed.
```

Now for the new definitions introduced as part of this extension, not just changes to the above already existing ones. We have a new synthesized attribute as well:

```
Inductive rep_on_RepMin : RepMin -> nat -> RepMin -> Prop :=
| rep_on_leaf : forall m gm, rep_on_RepMin (leaf m) gm (leaf gm)
| rep_on_inner : forall x x' y y' gm,
                 rep_on_RepMin x gm x' ->
                 rep_on_RepMin y gm y' ->
                 rep_on_RepMin (inner x y) gm (inner x' y')
| rep_on_fwd : forall t t_fwd t' gm,
               RepMin_fwds t t_fwd ->
               rep_on_RepMin t_fwd gm t' ->
               rep_on_RepMin t gm t'
.
```

Notably, this attribute has an inherited attribute parameter. The inherited equations for each child show up in how this synthesized attribute is "invoked" on each child. In this case, it's just copying down the same value for `gm`.

Note also in the above evaluation relation, we do not have an explicit equation for `three` in the original attribute grammar. As a result, we write a generic evaluation via forwarding copy equation (`rep_on_fwd`).

Finally, we have a new inductive relation and property to prove, which with one exception does not show us anything special.

```
Inductive eq_all : RepMin -> nat -> Prop :=
| eq_leaf : forall m, eq_all (leaf m) m
| eq_inner : forall x y m,
             eq_all x m ->
             eq_all y m ->
             eq_all (inner x y) m
| eq_three : forall x y z m,
             eq_all x m ->
             eq_all y m ->
             eq_all z m ->
             eq_all (inner x y) m
.

Theorem rep_all_equal :
  forall t m r,
  rep_on_RepMin t m r ->
  eq_all r m.
Proof.
  intros t m r SYN. induction SYN.
  (* Leaf case: directly by construction *)
  constructor.
  (* inner cast: directly, using the induction hypotheses *)
  constructor; assumption.
  (* fwd case: apply induction hypothesis *)
  apply IHSYN.
Qed.
```

The only interesting aspect to note is the proof goal in the forwarding case:

```
H : RepMin_fwds t t_fwd
```

```
SYN : rep_on_RepMin t_fwd gm t'
IHSYN : eq_all t' gm
______________________________________(1/1)
eq_all t' gm
```

This is the kind of utility that induction on decorated trees (in this case, on the evaluation relation of this synthesized attribute) has. This case is quite trivial to show, because we're just copying the value from the forwarding tree, which our induction hypothesis already makes claims of.

A.3 Coherence of RepMin for section 6.4

Continuing from the previous section, let us show the original property is coherent still in the extended language.

First, we define a coherent property (specialized to RepMin):

```
Definition coherent P :=
  forall t t_fwd, RepMin_fwds t t_fwd -> (P t <-> P t_fwd).
```

Note this is a property about a property (about RepMin trees). One important (sometimes) difference: we're using `RepMin_fwds` which is just the evaluation relation for forwarding. Notably, this is not exactly the same as our $t.forward$ function, which indicates that $t.forward = t$ for trees rooted in non-forwarding productions. However, for the examples we show below, this makes no difference except reducing tedium by making those trivial cases impossible.

First, we show coherence of the relations we defined. These are binary relations, but the non-tree parameter doesn't really matter so we can brush it aside to only work with our coherent property definition above.

```
Theorem lte_all_coherent :
  forall n, coherent (fun t => lte_all t n).
Proof.
  unfold coherent. intros n t t_fwd FWD.
  induction FWD; intuition.
    (* Show three -> inner inner *)
      (* exploit definition of lte_three *)
      inversion H; subst.
      (* show lte inner inner *)
      constructor; try constructor; assumption.
    (* Show inner inner -> three *)
    inversion H; subst. inversion H2; subst.
    constructor;assumption.
Qed.

Theorem exis_coherent :
  forall n, coherent (fun t => exis t n).
Proof.
  unfold coherent. intros n t t_fwd FWD.
  induction FWD; intuition.
  (* three -> inner inner *)
  inversion H; subst.
    (* Tedious case selection: either x y or z *)
    inversion H4. constructor. left. constructor. left. assumption.
    inversion H0. constructor. left. constructor. right. assumption.
    constructor. right. assumption.
  (* inner inner -> three *)
  constructor. inversion H; subst.
  inversion H3. inversion H0; subst. inversion H5.
    (* More case selection *)
    left. assumption.
    right. left. assumption.
    right. right. assumption.
Qed.
```

Since our evaluation relation for `min` computes the same value on forwarding productions as what it forwards to, we can simply show the coherence of the evaluation relation directly:

```
Theorem min_coherent :
  forall m, coherent (fun t => min_on_RepMin t m).
Proof.
  unfold coherent. intros m t t_fwd FWD.
  induction FWD; intuition.
    (* three -> inner inner: exploit def of min_three *)
    inversion H; subst.
    (* now just apply def on min_inner *)
    constructor; try constructor; assumption.
    (* inner inner -> three: exploit def of min_inner *)
    inversion H; subst. inversion H2; subst.
    constructor; assumption.
Qed.
```

Finally, our task is to show the coherence of the property. This proof is supposed to be simple, and it very much is. We don't even need to actually show the kinds of lemmas we discuss in the chapter (the closure properties for coherence), as they are such simple lemmas they're taken care of automatically by the Coq `intuition` tactic in this case. All we need to do is introduce the above coherence properties for each involved relation.

```
Theorem mc_coherent :
 forall m, coherent (fun t =>
           min_on_RepMin t m ->
           lte_all t m /\ exis t m).
Proof.
  unfold coherent. intros m t t_fwd FWD.
  induction FWD.
    (* A convenience: remember the fwding relationship *)
    assert (F : RepMin_fwds (three x y z) (inner (inner x y) z)).
      constructor.
    (* exploit min_coherent to see three and inner inner give same m *)
    specialize (min_coherent m (three x y z) (inner (inner x y) z) F);
      intros.
    (* exploit lte_all_coherent *)
```

```
   specialize (lte_all_coherent m (three x y z) (inner (inner x y) z) F);
     intros.
    (* exploit exis_coherent *)
    specialize (exis_coherent m (three x y z) (inner (inner x y) z) F);
     intros.

    (* And then just apply these lemmas *)
    intuition.
Qed.
```

There is more to showing non-interference. After showing our properties are coherent, we need to show that our attribute grammar preserves all coherent properties. However, this sort of proof is extraordinarily difficult to write in Coq. The trouble is that we need to show that something that *was* coherent is still coherent after we have made changes to the attribute grammar (i.e. added forwarding productions). This kind of "what happens when I make changes to a type" is difficult to express. Meanwhile, the development in the chapter is rather intuitive (the "unreasonable" approach), so we will stop here.