Reflection of Terms in Attribute Grammars: Design and Applications

Lucas Kramer, Ted Kaminski, Eric Van Wyk
Department of Computer Science & Engineering
University of Minnesota
Minneapolis, Minnesota, USA
krame505@umn.edu, tedinski@cs.umn.edu, evw@umn.edu

Abstract

This paper shows how reflection on undecorated syntax trees used in attribute grammars, that is, terms, can significantly reduce the amount of boiler-plate specifications that must be written. The proposed reflection system is implemented in the form of a `reflect` function mapping terms and other values into a generic representation and a `reify` function for the inverse mapping. The system is implemented in the Silver attribute grammar system. We demonstrate the usefulness of this approach to reflection in attribute grammars in several ways. The first use is in the serialization and de-serialization of the interface files Silver generates to support separate compilation; a custom interface language was replaced by a generic reflection-based implementation. Secondly, we describe an extension to Silver itself that simplifies writing language extensions for the ableC extensible C specification by allowing language engineers to specify C-language syntax trees using the concrete syntax of C (with typed holes) instead of writing abstract syntax trees. Third, strategic term rewriting in the style of Stratego is implemented using reflection as a library for, and extension to, Silver. Finally, an experimental implementation of staged interpreters for a small staged functional language is discussed.

1. Introduction

Strong static type systems are a lightweight, yet effective, kind of formal methods for ensuring that run-time type errors cannot happen when executing type-correct programs. Many find the benefits of type safety outweigh the restrictions that such systems necessarily impose. Some meta-programming languages and systems enjoy the benefits of strong static typing, *e.g.* the Kiama [1] system embedded in Scala, the Java-based JastAdd [2] attribute grammar system, and the Silver [3] attribute grammar system used in the work presented here. The type systems in these ensure that all object-language syntax trees are well-sorted, that is, they correspond to the context-free grammar defining the abstract syntax of the language being specified by these systems.

But as many have observed, for example in Lämmel and Peyton Jones's "scrap-your-boilerplate" work [4], well-sortedness often comes at a price; transformations over syntax trees from syntactically rich languages are cumbersome to express. For example, consider implementing a program transformation that rewrites $x + 0$ to x; a recursive implementation typically requires code or specifications not only for the case of addition, but also for all other constructs in the language. For these other constructs the specification simply duplicates each subtree with its rewritten components, leading to a large amount of uninteresting, cumbersome boilerplate code.

One approach to this problem is to use a form of *reflection* [5]. In the approach presented here, a (well-sorted) term is reflected into a generic form over which transformations such as the ones described above can be dramatically simplified. In the example uses discussed here there are cases in which thousands of lines of language specifications can be eliminated, or did not otherwise have to be written, due to the use of the proposed reflection system. The generic form used here has just a few mechanisms for representing all terms, regardless of the constructor (grammar production) used to build it. Thus the code written to, for example, serialize a tree need only be written for the few cases in this generic form. This ease of writing is traded for the loss of type-safety as well-sortedness is not guaranteed when constructing or manipulating object-language trees in this form. This paper presents a technique for bringing reflection of terms (syntax trees without attributes) to attribute grammars and evaluates it on several examples.

1.1. Motivating Example

The reflection system presented here is implemented in Silver and some example uses are in an extensible specification of C, ableC [6], all of which are written in Silver. As a first example, consider a language extension that introduces an exponent operator `x ** y` to C; this new expression should translate to the C code shown in Figure 1. Directly constructing the syntax tree of this translation by calling abstract productions is quite tedious and imposes barriers to entry for new language developers who must learn the numerous productions in the ableC abstract syntax grammar to become productive. Instead it is desirable to extend the meta-language, in this case Silver, to allow abstract syntax trees to be constructed using the actual concrete syntax of the object-language, as can be done in systems such as Stratego [7].

```
1  ({ float _res = 1;
2     for (int _i = 0; _i < y; _i++) {
3       _res *= x;
4     }
5     _res; })
```

Figure 1: The translation of an exponent expression `x ** y`, where `x` is a `float` and `y` is an `int`. Note this uses a statement-expression `({...; ...;})`, a C extension supported by GCC and other compilers, to embed a statement in an expression.

An example use of object-language concrete syntax is shown in Figure 2. Here the extension developer introduces a new abstract syntax production (named `exponent`) for an exponent operator that extends the ableC host language, and specifies that this new exponent construct translates down (via the `forwards to` specification [8]) to a plain C language abstract syntax tree like the one in Figure 1. Some attributes (such as pp, the "pretty-printed" version of the construct) may be defined directly on the new production, while all other attributes (such as an attribute to translate the program to some executable form) are automatically computed on the forwarded-to tree.

Concrete syntax of the object language, in this case C, is embedded here in Figure 2 as an expression in Silver using a so-called *quote* production that wraps a piece of code in the object-language (C) for which the abstract syntax tree should be constructed; `ableC_Expr {...}` beginning on line 6 of Figure 2 is an instance of a quote production that introduces an `Expr` term from the C language. Sometimes portions of the desired tree are not fixed but instead should be constructed using Silver code. Such holes in the tree may be specified in using so-called *antiquote* productions that escape from the object-language syntax back to Silver expressions, such as `$TypeExpr` on line 7. This line specifies the declaration and initialization of `_res` seen on line 1 of Figure 1. The specification in Figure 3 is what is needed when writing the abstract syntax directly to specify this same single line. Since Silver itself is an extensible language, language developers may easily construct a version of Silver tailored to their object-language (such as the ableC extension to Silver used here, referred to as Silver-ableC) by introducing appropriate quote and antiquote productions.

Computing the translation of quote productions (such as `ableC_Expr`) into ordinary Silver expressions presents a difficulty without a reflection system like the one presented here; we must generate the abstract syntax for the

```
1  abstract production exponent
2  top::Expr ::= base::Expr exp::Expr
3  {
4    top.pp = base.pp ++ " ** " ++ exp.pp;
5    forwards to
6      ableC_Expr {
7        ({ $TypeExpr{base.ty.tyExpr} _res = 1;
8           for ($TypeExpr{exp.ty.tyExpr} _i = 0; _i < $Expr{exp}; _i++) {
9             _res *= $Expr{base};
10          }
11          _res; })
12     };
13 }
```

Figure 2: An example using Silver-ableC to implement an exponent expression in ableC.

```
1  declStmt(
2    variableDecls(nilStorageClass(), base.ty.tyExpr,
3      consDeclarator(
4        declarator(name("_res"), baseTypeExpr(),
5          justInitializer(
6            exprInitializer(integerConstant("1", false, noSuffix()))),
7        nilDeclarator())))
```

Figure 3: The tedious code that would be written using only plain SILVER, for just line 7 of the exponent production in Figure 2.

SILVER expression which, when compiled and evaluated, will construct the specified object-language syntax tree. In systems like STRATEGO this process may be accomplished by the use of rewrite rules; this is permitted as terms have a uniform/generic representation, and weak typing allows for an incremental transformation of object-language into meta-language abstract syntax. However in a strongly typed system such as SILVER, object-language trees are strongly distinguished from meta-language expressions that construct trees; thus a direct transformation is required from the former to the latter.

One approach is to define a translation-to-SILVER attribute on all nonterminals of the object-language attribute grammar, on each production writing an equation that constructs the SILVER expression for a call to that C-language production. With nearly 500 abstract productions in the ABLEC specification doing this would require writing a tremendous amount of boilerplate code. An important observation is that the desired translation does not depend on the semantics of particular productions, but rather is only based on the name of the production and number of children. Thus, some approach is needed for dealing with a generic representation of an abstract syntax tree, rather than performing a computation on the well-sorted tree itself.

There are other similar problems that can best be expressed as generic analyses on trees. For example consider serializing and de-serializing between a term and a text string; this can easily be done with a uniform untyped term representation, but not so easily with a well-sorted tree using a variety of nonterminals. Thus we want a general-purpose mechanism that can represent a term in a generic way, and convert between well-sorted terms and this generic representation.

1.2. Reflection in SILVER

Problems of like those described above can benefit from some form of reflection in the meta-language. Here we present a reflection mechanism, implemented in SILVER, consisting of two primary operations:

- **`reflect`**: which converts a well-sorted terms (syntax trees without attributes) in SILVER into a generic tree representation of type `AST`, defined below. This type is a nonterminal in a small SILVER grammar with a small set of productions for representing terms in a generic manner.
- **`reify`**: which converts generic `AST` trees back to their original well-sorted term form. This requires run-time type checking and may fail if the generic form does not correspond to a well-sorted tree in the object-language. Thus **`reify`** returns a value of type `Either<String a>`: either a string error message or a tree of the correct type (represented by type variable `a`).

Transformations and analyses on `AST` trees are implemented by specifying attributes and their defining equations. The resulting specifications that are much less verbose with much less boilerplate code than the equivalent specifications on well-sorted trees.

These reflection features can be used to solve the object-language to meta-language translation problem in a significantly more concise manner. Instead of computing the translation directly using attributes on the object-language syntax tree, we can **`reflect`** this tree into the `AST` representation, and compute the desired translation on this `AST` tree. This allows us to view data, that is syntax trees, in two ways. The first is through SILVER's type system as well-sorted terms. The second view is a more generic one in which values of different types or sorts can be viewed in a uniform way by the **`reflect`** transformation into a generic `AST` representation.

1.3. Contributions

The primary contributions of the paper are enumerated below.

- We integrate reflection and attribute grammars using a special `AST` nonterminal and bidirectional **`reflect`**/**`reify`** transformations between `AST` trees and well-sorted object-language syntax terms; the design of this system is discussed in Section 3 and its implementation in Section 9.
- We use reflection in conjunction with Silver's attribute grammar and parser specification features to provide a serialization/de-serialization library for arbitrary trees. This allowed 1,698 lines of hand-coded environment tree serialization/de-serialization code in the Silver compiler to be replaced with only 257 lines, removing 3.75% of the Silver specification. (Section 4)
- We demonstrate the use of reflection in mapping object-language concrete syntax to the Silver constructs that would construct it, allowing specifications like those in Figure 2. This saved almost 18,000 lines of specification over several ableC extension specifications, amounting to 40% of the code base by character count. The code generation for many extensions was complex enough that they would likely not have been attempted had only the direct method of specifying translation (as shown in Figure 3) been available. (Section 5)
- We demonstrate the use of reflection in attribute grammars to implement a strategic term rewriting library and language extension in the style of Stratego [7]. One use of this system is to perform "scrap your boilerplate"-style [4] substitutions of particular subtrees within much larger abstract syntax trees. This saved over 2,500 lines of specification between ableC and several extensions, amounting to 11.8% of the ableC specification code base. (Section 6)
- We implement an interpreter for a simple staged language as an attribute grammar specification using reflection. (Section 7)

This paper is an extension and revision of the paper "Reflection in Attribute Grammars" by the same authors[9], presented at the 2019 ACM SIGPLAN International Conference on Generative Programming: Concepts & Experiences (GPCE). This paper extends and expands the discussion in that paper of the reflection system and its use in the serialization/de-serialization example, the object-language concrete syntax example, and the staged language example listed above. The most significant change is the addition of a new example for a Stratego-inspired term rewriting system found in Section 6. This replaces an example of a specialized substitution mechanism used for instantiating C++-style templates in ableC. The term rewriting extension to Silver is used to implement substitution as well, but is more general. Examples of this term rewriting extension include the $x + 0 \to x$ optimization and an implementation of a λ-calculus evaluator based on an implementation in Stratego [10]. This section discusses the features in the Silver library that implements term rewriting and the new syntax provided by an extension to Silver that simplifies the use of this library.

Silver version 0.4.3[1] [3] is the attribute grammar system used in this paper. The ableC[2] [6] code and extensions to it [11] are from version 0.1.5 and the Silver-ableC[3] code is from version 0.1.3. Specifications from each are shown in several figures below and can be found in `grammars` directory of the corresponding repository in the directory or file name given in caption in each figure. A description of how the number of lines saved in Silver specifications or `ableC` extension specifications by using reflection was computed is given in the relevant section. New keywords introduced by extensions we discuss here are highlighted in the figures as **`bold`**.

[1]Available at `http://melt.cs.umn.edu/silver` and `https://github.com/melt-umn/silver`, archived at `https://doi.org/10.13020/D6QX07`.

[2]Available at `http://melt.cs.umn.edu/ableC` and `https://github.com/melt-umn/ableC`, archived at `https://doi.org/10.13020/D6VQ25`.

[3]Available at `http://melt.cs.umn.edu/` and `https://github.com/melt-umn/silver-ableC`, archived at `https://doi.org/10.13020/hbr0-9z50`.

```
1  nonterminal Either<a b>;
2
3  synthesized attribute isLeft :: Boolean occurs on Either<a b>;
4  synthesized attribute fromLeft<a> :: a occurs on Either<a b>;
5
6  abstract production left   top::Either<a b> ::= value::a
7  { top.fromLeft = value;
8    top.isLeft = true;
9  }
10
11 abstract production right  top::Either<a b> ::= value::b
12 { top.fromLeft = error("fromLeft accessed on a right instance of Either");
13   top.isLeft = false;
14 }
```

Figure 4: Some sample Silver specifications, these defining a polymorphic nonterminal `Either`, and productions and attributes for it.

2. Background: Attribute Grammars and Silver

In this section we provide some background on attribute grammars (AGs) and the Silver [3] attribute grammar system since reflection is integrated into this particular system. Essentially, attribute grammars provide a "semantics of context-free languages" [12] by associating semantic attributes (semantic values) with nodes in a well-sorted syntax tree. These decorated/attributed trees conform to a context-free grammar that specifies the (abstract) syntax of the language. These attributes carry information (semantics) about that language construct. For example, consider an attribute grammar for type checking a functional language. A nonterminal for expressions may be decorated with an attribute that specifies the type of the expression and another providing a typing context that maps names to their types. Equations are associated with the grammar productions and are used to determine the values of these attributes.

More formally, we may define an attribute grammar as a tuple

$$AG = (G, A, O, \Gamma, E)$$

where G is a context-free grammar (NT, T, P) with a finite set of nonterminal symbols NT, a finite set of terminal symbols T which includes basic types for integers and strings, and P a set of grammar productions. Grammar productions have the form

$$n_0 :: NT_0 ::= n_1 :: X_1 \ \ldots \ n_k :: X_k$$

Productions have a left-hand side nonterminal NT_0 and 0 or more right-hand side symbols in $NT \cup T$. Symbols in productions are are labeled (*e.g.* n_0, n_i) so that tree nodes can be easily referenced when defining or accessing attribute values on them. In Silver, productions are labeled as `concrete` if they are passed to the Copper [13] scanner and parser generator bundled with Silver. Those labeled by `abstract` are not; they are used to define the abstract syntax of a language and are our primary focus here, as seen in Figure 2.

Nonterminals and productions are analogous to, and often used like, inductive data types and their constructors as commonly found in functional languages such as ML and Haskell. Silver also has primitive types, including `Integer`, `String`, and `Boolean`, and a built in notion of lists, all of which can also be used on the right hand side of abstract productions. Hindley-Milner-style parametric polymorphism is also supported in Silver and type variables are written using lowercase names. An example of using attribute grammars in Silver in this way can be seen in Figure 4. Line 1 declares a nonterminal that has two type parameters. Lines 6 and 11 declare two abstract productions, one named `left` and one named `right`, for constructing trees of type `Either`. Note that `top` is commonly used as the name of the tree being constructed but this is not required. Both productions here have a single right-hand side element named `value` of the appropriate type parameter.

In an attribute grammar (G, A, O, Γ, E), the set A is the set of attributes and is partitioned into synthesized attributes A_S that propagate information up the syntax tree and inherited attributes A_I that propagate information down

the tree. The relation O is called an occurrence relation and it indicates which attributes decorate which nonterminals. Attributes can hold various types of values beyond primitive types such as strings and integers. Vogt et. al. [14] introduced higher-order attributes in which attributes contain terms, that is (yet undecorated) syntax trees. Attribute can also hold (references to) decorated syntax trees. Reference attributes [15] and remote attributes [16] allow decorated trees to be passed around as attribute values; these can be thought of as references or pointers to remote nodes in the syntax tree. In Figure 4, line 3 declares a Boolean synthesized attribute named `isLeft` that occurs on `Either` nonterminals to indicate if a tree is constructed by the `left` production. The attribute `fromLeft` can be used to access the value provided in the `left` production in the case that that production was used to construct the `Either` tree. This attribute is parameterized such that on a tree of type `Either<String Integer>`, the `fromLeft` attribute will have type `String`. Similar `isRight` and `fromRight` attributes are also defined.

The types of values to be held in attributes and the types/signature of productions is tracked by a typing context Γ. This is used to type check the equations in E that specify the values of attributes in a syntax tree. Equations in E are associated with a single production $p \in P$ and typically have the form $n_i.a = e$ with n_i being a label for a symbol in p. If n_i labels the nonterminal on the left hand side of the production then a must be a synthesized attribute used to compute a value decorating n_0. Otherwise a must be an inherited attribute computing a value for a child of n_0. Examples of equations can be seen in Figure 4. Line 7 defines `fromLeft` to be `value`, and line 8 defines `isLeft` to be `false`. An attribute grammar evaluator is used to find a solution to the equations in E for a particular syntax tree. There are various ways to implement such an evaluator, for example, techniques for ordered attribute grammars [17] or the demand-driven approach [18] used in SILVER and JASTADD.

Expressions on the right hand side of equations support many expected constructs, such as conditional expressions and production applications for building trees. SILVER also support pattern matching as an alternative to using attributes such `isLeft` and `fromLeft` as seen above. SILVER makes a distinction between trees that are decorated with attributes and those that are not; the `new` operator is used to extract an undecorated term from a decorated syntax tree.

Equations are typically associated with a production by writing them with the definition of the production, as is done for the equation for the pretty-print attribute pp on line 4 of Figure 2. But equations can also be written separately from their production definition in what are called *aspect* productions, as originally specified in JASTADD. These are labeled with `aspect` and allow new attribute equations to be added for existing productions in a language that is being extended.

Another attribute grammar feature in SILVER is *forwarding* [8]. Forwarding allows a production like the `exponent` production on line 5 in Figure 2 to specify a syntax tree that it is semantically equivalent to. Here, that is the tree with the for-loop that computes the value of the exponent. If the forwarding tree (the one constructed by the production with a `forwards to` clause) has an equation for an attribute, then the value determined by that equation is used. If there is a query for an attribute for which the forwarding tree does not have a defining equation, then the forwarded-to tree is computed and decorated and the attribute value from that tree is used instead. This is used extensively in extensible language specifications and in some examples below. It allows language extensions to define attributes that have values specialized to the extension that differ from the values of the same attributes on the forwarded-to tree. This is useful in ABLEC [6], for example, in an extension that introduces inductive/algebraic data types to C since type errors in the use of these data types (either in their construction or in pattern matching on them) can be reported in terms of the extension and not in terms of the generated C code.

Another SILVER feature used below is *collection production attributes*; these are local attributes defined within a production. These hold values defined within production bodies with an initial value and an append operator for values of that type (*e.g.* for lists this is the empty list `[]` and list append operator `++`.) Aspects of a production containing a collection may contribute additional values, to be incorporated using the append operator.

In SILVER, names of productions and attributes have fully-quantified forms that include the name of the grammar in which they were defined, a bit like fully qualified names in Java (but using colons instead of dots to separate names). The specification in Figure 4 is in the `core` grammar, so the fully qualified name of `left` is `core:left`. Similarly, `ableC:addExpr` is the `addExpr` production in the `ableC` grammar. For reasons of brevity, we use `ableC` as the grammar name here but the actual grammar name is `edu:umn:cs:melt:ableC:abstractsyntax:host`, as can be seen in the specifications in the ABLEC repository[2]. Finally, note that when the context permits we will also refer to grammar elements using their shorter un-qualified name.

3. Design of the Reflection System

In this section we describe the design of the reflection features and how they are used in attribute grammars. The implementation is discussed later in Section 9.

We first introduce the type AST; this is a generic representation for well-sorted terms. There are two operations over these: **reflect** which transforms well-sorted syntax trees into generic AST values, and **reify**, which maps generic trees back into well-sorted terms. The AST type is defined as a nonterminal in a Silver library, shown in Figure 5 on line 3. AST trees may be constructed in the same way as other trees in Silver through the application of productions to primitive values or other trees. Each AST production in Figure 5 corresponds to a type (or class of types) in the Silver language: *e.g.* stringAST to String, nonterminalAST to nonterminals (well-sorted terms), terminalAST to terminal symbols (tokens returned from a scanner, occurring in concrete syntax trees), and listAST to Silver lists. Trees built by the polymorphic anyAST production contain values (represented by the type variable a) that do not typically occur in abstract or concrete syntax trees: these includes functions which we cannot directly inspect, and references to decorated trees which may be circular. The ASTs nonterminal encodes a sequence of AST trees, such as in the children of a production or a list value.

The **reflect** operation uses these productions to produce an AST value from any value and thus has the type AST ::= a. (A more familiar writing of this type might be a -> AST, but functions in Silver use the same type ::= type notation as productions.) For example, **reflect** will transform the Silver tree

```
integerConstant("42", true, noIntSuffix())
```

on line 6 of Figure 3 into the AST tree

```
nonterminalAST("integerConstant",
  consAST(stringAST("42"),
    consAST(booleanAST(true),
      consAST(
        nonterminalAST(
          "noIntSuffix", nilAST()),
        nilAST())))))
```

```
1  grammar silver:reflection;
2
3  nonterminal AST;
4  abstract production nonterminalAST
5  AST ::= prodName::String
6    children::ASTs
7
8  abstract production terminalAST
9  AST ::= terminalName::String
10   lexeme::String location::Location
11
12 abstract production listAST
13 AST ::= vals::ASTs
14
15 abstract production stringAST
16 AST ::= String
17
18 abstract production integerAST
19 AST ::= Integer
20
21 abstract production floatAST
22 AST ::= Float
23
24 abstract production booleanAST
25 AST ::= Boolean
26
27 abstract production anyAST
28 AST ::= a
29
30 nonterminal ASTs;
31 abstract production consAST
32 ASTs ::= h::AST t::ASTs
33
34 abstract production nilAST
35 ASTs ::=
```

Figure 5: The Silver nonterminals and productions used to represent abstract syntax trees. Production bodies, which contain attribute equations, are not shown. See core/reflect/AST.sv

As with all nonterminals, attributes may be associated with the AST nonterminal. Aspect productions then specify new equations for the abstract productions in Figure 5; this is typical for applications using the reflection system. The Silver-ableC extension discussed in Section 1 introduces a new translation attribute on AST, and provides equations for this attribute on AST production to produce the Silver abstract syntax that is needed. This process is described in more detail in Section 5. Another aspect-defined attribute is serialize, used to print out an AST tree and discussed in Section 4.

In many of the applications discussed in the following sections a generic `AST` tree is converted back into a well-sorted object-language term by the **`reify`** operator. This has type `Either<String a> ::= AST`, taking an `AST` and returning either a `String` error message or the well-sorted term. Because `AST` trees can be constructed directly using the productions in Figure 5, it is possible to construct trees that do not correspond to a well-sorted tree. An attempt to reify such a tree will result in an error, returning `left(s)`, where `s` is an error message of type `String`. A successful reification wraps the tree of type `a` in the `right` constructor. The reflection system satisfies the following invariant for any tree `t`:

```
reify(reflect(t)) = right(t)
```

In our experience large `AST` trees are rarely constructed directly; they usually are constructed by **`reflect`**.[4]. Thus ill-sorted `AST`s (and thus failures when calling **`reify`**) do not occur in typical uses of the system. Even when an error does occur it is reported immediately by the call to **`reify`**, and we have found finding and fixing the source of such errors to be straightforward. We do, however, recognize that this might not be the case for all Silver users.

Despite this, during the reification process we must check that the `AST` in question actually corresponds to the inferred result type of **`reify`**. This requires a run-time type checking process, looking up productions and terminals to ensure they are defined and match the expected types. Since Silver supports polymorphic data types (*e.g.* nonterminals such as `Either<a b>`), it may not be possible to compute the final type of a sub-tree from only its arguments, so type checking also must perform Hindley-Milner type inference.

4. Reflection for Automatic Tree Serialization and De-serialization

The process of serializing well-sorted terms to strings and de-serializing strings back into terms is a common one that can involve a significant amount of boiler-plate code. This problem existed in the specification of the Silver language. In the Silver specification language, a grammar module may import other grammar modules in order to extend the language specified in another imported module. The Silver system supports separate compilation of grammar modules by maintaining an interface file for each module that describes the grammar elements, such as attributes, productions and nonterminals, that are declared and certain properties of them, such as their type. The Silver language is bootstrapped in Silver so there is a Silver specification of the Silver language. A significant source of boilerplate specification in this Silver specification is in its module for serializing and de-serializing syntax trees that represent the environment for a grammar stored in these interface files. Separate compilation of grammar modules uses the serialization of environments as interface files to avoid reading an unchanged source grammar imported by another grammar that is being compiled. Originally, serialization of environments was implemented by defining an `unparse` attribute on all environment nonterminals and their productions to construct a string representation, and de-serialization was handled by a parser using a rather large custom grammar.

Using reflection and `AST` trees we can define a generic implementation of this process, replacing 1,698 lines in the Silver compiler with 257 lines, and saving 1,441 out of 38,400 lines (3.75%) of the entire code base. [5] This change does come at a cost, as the generic interface files are on average 215% larger than before (measured for the ableC code base), less readable by humans, and their de-serialization adds approximately 2-3 seconds in total to build times taking around 60 seconds. However we believe such penalties are worth the significant savings in code complexity.

A visualization of the serialization/de-serialization process is shown in Figure 6. At the top is a simplified notion of an environment consisting of a list of three pairs, mapping `a` to `6`, `b` to `8`, and `c` to `7`. Serialization is done by reflecting a tree into an `AST`, the middle box of the figure, and accessing the `serialize` attribute on the `AST`, which then produces the string value at the bottom. This attribute, line 1 of Figure 7, is actually of type `Either<String String>`; this is because serialization may fail if a non-printable value, such as a function, is a component of the reflected tree constructed as an `anyAST` tree (line 21). The `left` side of `Either` encodes an error message and the `right` side the successful result, as seen before in the return type of **`reify`**. Integer and string `AST`s (lines 13 and 17) are serialized as expected, with string special characters needing to be escaped first.

[4] Transformations that involve directly constructing new `AST`s are less common but do occur, as will be demonstrated with the implementation of rewriting in Section 6. But even in this case, large portions of the final `AST` to be reified are copied unchanged from the initial `AST` returned by **`reflect`**.

[5] See `https://github.com/melt-umn/silver/pull/255`; note that not all changes here are related to serialization/de-serialization.

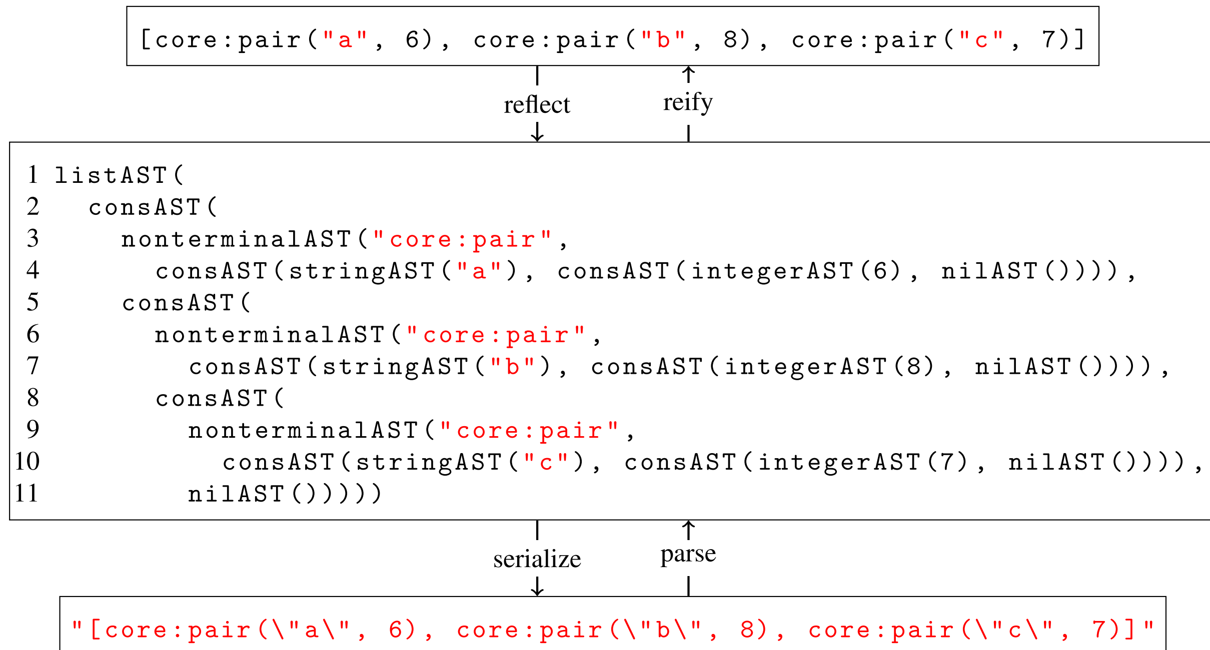

Figure 6: An example of serialization/de-serialization.

```
1  synthesized attribute serialize::Either<String String>
2    occurs on AST, ASTs;
3  aspect production nonterminalAST
4  top::AST ::= prodName::String children::ASTs
5  {
6    top.serialize =
7      case children.serialize of
8      | right(childSer) -> right(prodName ++ "(" ++ childSer ++ ")")
9      | left(msg) -> left(msg);
10     end;
11 }
12
13 aspect production integerAST
14 top::AST ::= i::Integer
15 { top.serialize = right(toString(i)); }
16
17 aspect production stringAST
18 top::AST ::= s::String
19 { top.serialize = right("\"" ++ escapeStr(s) ++ "\""); }
20
21 aspect production anyAST
22 top::AST ::= x::a
23 { top.serialize = left("Cannot serialize an anyAST"); }
```

Figure 7: Some of the SILVER attribute equations for serialization. See silver/reflect/AST.sv

```
grammar silver:serialization;

nonterminal AST_c with ast<AST>;
synthesized attribute ast<a>::a;

concrete productions top::AST_c
| n::QualifiedName_t '(' children::ASTs_c ')'
  { top.ast = nonterminalAST(n.ast, foldr(consAST, nilAST(), children.ast)); }
| n::QualifiedName_t '(' ')'
  { top.ast = nonterminalAST(n.ast, nilAST()); }
| '[' vals::ASTs_c ']'
  { top.ast = listAST(foldr(consAST, nilAST(), vals.ast)); }
| '[' ']'
  { top.ast = listAST(nilAST()); }
| s::String_t -- matches a quoted string literal, must be un-escaped:
  { top.ast =
      stringAST(unescapeString(substring(1, length(s.lexeme) - 1, s.lexeme))); }
| i::Int_t
  { top.ast = integerAST(toInteger(i.lexeme)); }
| f::Float_t
  { top.ast = floatAST(toFloat(f.lexeme)); }
| 'true'
  { top.ast = booleanAST(true); }
| 'false'
  { top.ast = booleanAST(false); }

nonterminal ASTs_c with ast<[AST]>;

concrete productions top::ASTs_c
| t::ASTs_c ',' h::AST_c
  { top.ast = t.ast ++ [h.ast]; }
| h::AST_c
  { top.ast = [h.ast]; }
```

Figure 8: The grammar used to construct the de-serialization parser. Terminal declarations are straightforward, and are omitted to save space.

Of interest is the serialization of `nonterminalAST` trees (line 6 of Figure 7). We first compute the serialization of the child nodes; if any of these fail we wish to pass along the failure message. Otherwise a string is constructed using the resulting child serialization, the production name, and appropriate parentheses.

For de-serialization, we use Silver's built-in declarative parser specification features to define concrete syntax matching the serialized strings, shown in Figure 8. This grammar defines a concrete syntax for `AST` trees using the `AST_c` and `ASTs_c` nonterminals. Both have productions that are used in building the `AST` parser. Both are also decorated by an `ast` attribute that, on `AST_c` nonterminals is an `AST` tree and on `ASTs_c` nonterminals is a list of `AST` tress, written `[AST]`. The generated parser constructs the concrete syntax trees and the equations then convert these concrete syntax trees into `AST` values. These `AST` trees are then reified to recreate an environment, as illustrated on the right of Figure 6. Together, these provide a concise and convenient mechanism for specifying serialization and de-serialization of Silver syntax trees.

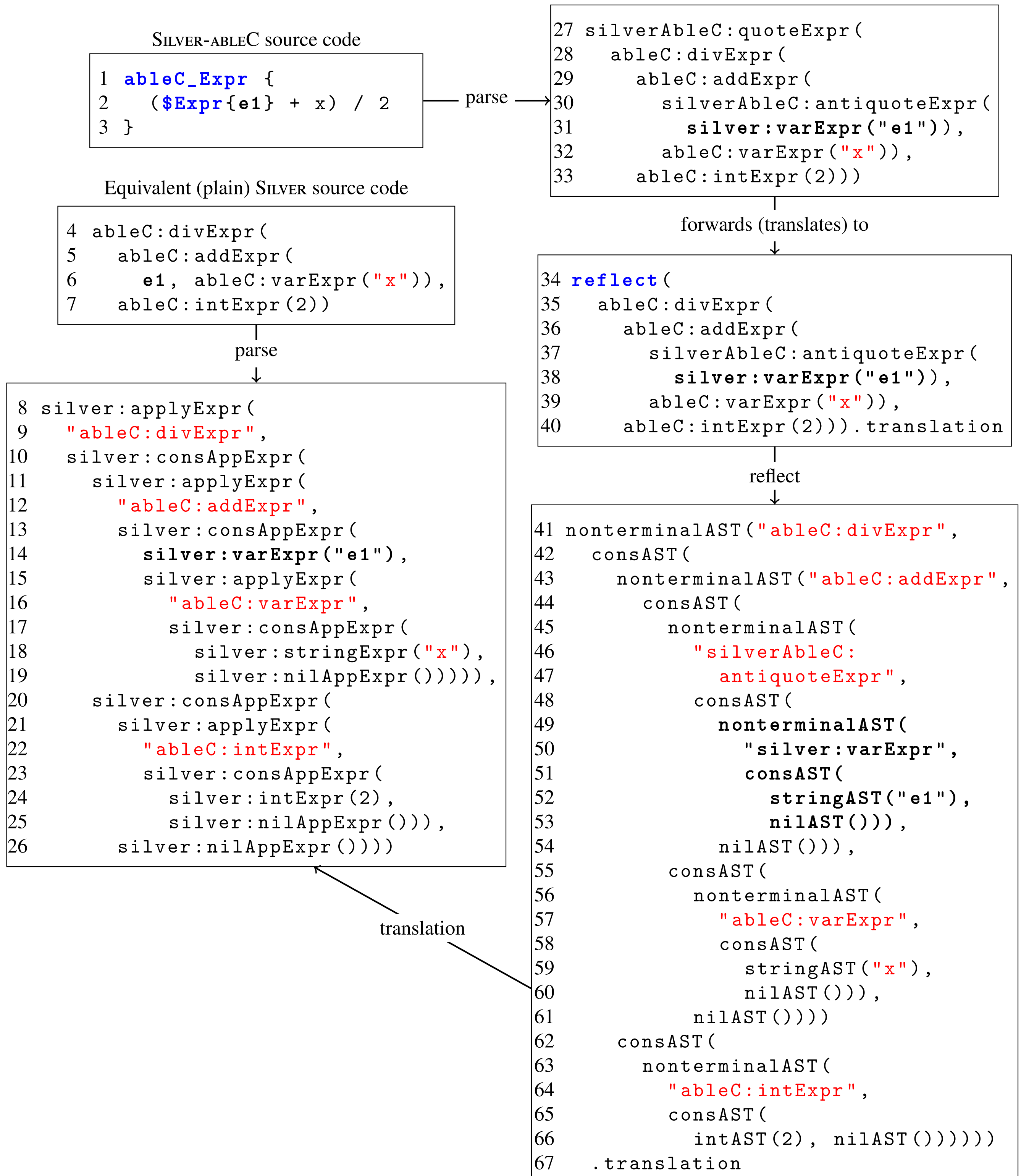

Figure 9: The translation process for an ABLEC quote production. In **bold** is shown an antiquoted piece of SILVER code; note that it is the same between the original parse result (top right) and the final translation abstract syntax tree (bottom left). Production and grammar names have been shortened for clarity.

5. Reflection for Tree Construction Using Object-Language Concrete Syntax

This section further describes the reflection system in SILVER and demonstrates its use in an extension to SILVER itself. This extension lets language developers specify trees in the object-language being developed using the concrete syntax of that language instead of its inconvenient abstract syntax, as briefly demonstrated in Section 1 by Figure 2. We demonstrate this on ABLEC, an extensible specification of C, defined in SILVER. The extension to SILVER for ABLEC is referred to as SILVER-ABLEC below. Any specification, including ABLEC, will have a large number of abstract productions and some, such as for type expressions in variable declarations, are complicated. Thus the ability to write concrete syntax instead of abstract syntax saves a tremendous amount of effort on the part of the language developer, and has led to the development of ABLEC extensions that we may not have even undertaken without this capability. The question we answer here is how reflection is used in translating the nice specification in Figure 2 into the implementation in Figure 3.

A step-by-step example use of SILVER-ABLEC is shown in Figure 9. At the top left is the code to construct a simple term written using SILVER-ABLEC, while beneath it is an equivalent piece of code written using plain SILVER (that is, without the SILVER-ABLEC extension.) Our goal is to translate the SILVER-ABLEC code into the same SILVER abstract syntax tree as would result from parsing the plain SILVER code. To keep things manageable, a simpler example than in Figure 2 is used to explain this process and the SILVER-ABLEC extension. The antiquoted expression `e1` (line 2) is written in bold and we track its translation through both processes of using and not using SILVER-ABLEC. The name `e1` will hold an ABLEC tree that is to be plugged into the expression, such that when the generated C code is executed the value of this expression will be added to the C variable `x` and divided by 2. This is a shorter example, but qualitatively the same as Figure 2.

Below this is the equivalent SILVER source specification that does not use the SILVER-ABLEC extension and requires one to write out the abstract syntax explicitly. This is a shorter example of the same thing as shown in Figure 3. Both the top-left and middle-left code fragments are functionally equivalent specifications and both need to translated into the same thing — a SILVER abstract syntax tree.

In the plain SILVER specification we see that division is represented by the ABLEC production `ableC:divExpr` (line 4) and the name `e1` (line 6) is the first argument to the addition production `ableC:addExpr` (line 5) This is simply parsed by the SILVER compiler to generate the abstract syntax representation in the lower left of Figure 9. Since SILVER is bootstrapped in SILVER the abstract syntax of a SILVER specification implemented as a SILVER attribute grammar. Thus the resulting abstract syntax is mostly applications of various SILVER productions; the production `silver:applyExpr` represents the application of productions (tree creation). We see a few instances of `silver:applyExpr` production with the first argument being the name of the production in the object-language, in this case ABLEC, and thus the names as strings are those fully qualified names from the ABLEC grammar. The string `"ableC:divExpr"` is seen on line 9 as this use of `applyExpr` represents the application of this ABLEC production for division. The second argument to `applyExpr` is the `AppExprs` list (constructed by `silver:consAppExpr` and `silver:nilAppExpr` productions) of trees to become the children of the constructed tree representing division: the first element being the representation of the addition, the second the constant 2. We see on line 14 the name `e1` used as a SILVER variable reference form of expression. From an abstract syntax tree like this, the SILVER compiler will type check the specification and generate the Java code that forms the attribute grammar evaluator [3].

5.1. Using Reflection

We now consider the other path to this abstract syntax tree, using C concrete syntax in the SILVER-ABLEC extension and the SILVER reflection system. The first step is again parsing, but now using the SILVER-ABLEC parser. This yields a SILVER tree containing quote (line 27) and antiquote (line 30) productions in the upper right box of Figure 9. The `quoteExpr` on line 27 contains the ABLEC abstract syntax that was written out directly in the plain SILVER specification. The SILVER-ABLEC parser is a combination of the concrete syntax for SILVER and ABLEC with syntax for the quote and antiquote productions on different ABLEC nonterminals (such as `ableC:Expr`). The antiquote production, parsed from **`$Expr`**`{e1}`, switches back into SILVER abstract syntax. Thus line 31 here matches line 14 in the tree in the lower left.

The definitions of a few of the quote and antiquote productions introduced by SILVER-ABLEC are shown in Figure 10. Quote productions such as `quoteExpr`, being extensions to the SILVER language, must specify the equivalent `silver:Expr` that they translate down to. This is done via forwarding; one can read "`forwards to`" as "translates

```
1  grammar silverAbleC;
2  imports silver, ableC;
3  imports silver:reflection, silver:metatranslation;
4
5  abstract production quoteExpr
6  top::silver:Expr ::= e::ableC:Expr
7  { forwards to reflect(new(e)).translation; }
8
9  abstract production antiquoteExpr
10 top::ableC:Expr ::= e::silver:Expr
11 {  }
12
13 aspect production nonterminalAST
14 top::AST ::= prodName::String children::ASTs
15 {
16   antiquoteProds <-
17     ["silverAbleC:antiquoteExpr", "silverAbleC:antiquoteStmt", ...];
18 }
```

Figure 10: A sampling of SILVER-ABLEC quote and antiquote productions, and the code to specify which productions are to be translated as antiquote productions rather than quoted ABLEC abstract syntax. See edu.umn.cs.melt.exts.silver.ableC/abstractsyntax

to".[6] This translation is the result of applying **reflect** to (a new undecorated copy of) the syntax tree `e` converting the `ableC:Expr` into a tree of type `AST`. (If we reflected the syntax tree `e` decorated with attributes, **reflect** would return an `anyAST` value.) These can be seen in the lower right of Figure 9; all productions (including antiquote ones) have been expanded into their `nonterminalAST` counterparts: `ableC:divExpr` on line 35 becomes `nonterminalAST("ableC:divExpr", ...)` on line 41.

From the reflected `AST`, we access the `translation` attribute (written `.translation`), seen on line 40 in the middle-right and line 67 in the lower-right of Figure 9. This attribute constructs the SILVER abstract syntax tree in the lower left, where ABLEC productions have been converted into `silver:Expr` trees build by `applyExpr` as discussed above. The same is true for the contents of antiquote productions which have been reified directly into `silver:Exprs`. Note how the antiquoted SILVER tree in the initial SILVER-ABLEC abstract syntax, shown in bold, has been preserved unchanged in the final abstract syntax tree.

Some equations for `translation` are shown in Figure 11. Every kind of SILVER value represented by `AST` has a straightforward transformation into a piece of SILVER code, for example an integer value (`integerAST` on line 7) is translated to a SILVER integer: `silver:intExpr(i)` on line 9. Here an aspect production is defining the `translation` attribute on a production defined in the `silver:reflection` grammar (line 18 of Figure 5). An attempt to translate back to SILVER an `anyAST` tree (line 9) results in an error being raised as these are non-inspectable values that can be reified, but little else.

However, there is a complication introduced when dealing with antiquote productions: they contain a piece of SILVER code that should be *evaluated* to obtain a tree and thus their translation should just be the wrapped piece of code. Unfortunately, this `silver:Expr` tree is no longer directly available as it has been "accidentally" reflected and turned into an `AST`. So when an antiquote production is encountered by the `translation` attribute, the child `AST` must be reified back into the `silver:Expr` that was originally specified.

When translating a `nonterminalAST`, some method of identifying whether a production is an antiquote production is required. This is done by way of a *collection production attribute* (line 18). The attribute `antiquoteProds`

[6] Conversely, antiquote productions are effectively extensions to the object-language (as opposed to quote productions that extend the SILVER meta-language.) To satisfy the requirements of the modular well-definedness analysis [19] they should forward to a translation in C. However, antiquote productions have no semantic equivalent in the object-language, and productions within a quoted tree will never have attributes demanded. Thus they forward (not shown) to a dummy value that raises an error if it is erroneously evaluated.

```
1  grammar silver:metatranslation;
2  imports silver, silver:reflection;
3
4  synthesized attribute translation<a>::a;
5  attribute translation<silver:Expr> occurs on AST;
6
7  aspect production integerAST
8  top::AST ::= i::Integer
9  { top.translation = silver:intExpr(i); }
10
11 aspect production anyAST
12 top::AST ::= x::a
13 { top.translation = error("anyAST error"); }
14
15 aspect production nonterminalAST
16 top::AST ::= prodName::String children::ASTs
17 {
18   collection antiquoteProds::[String] with ++;
19   antiquoteProds := [];
20
21   top.translation =
22    if contains(prodName, antiquoteProds)
23    then case children of
24           | consAST(a, nilAST()) -> reify(a)
25           | _ -> error("Expected only one child")
26           end
27    else silver:applyExpr(
28               silver:varExpr(prodName),
29               children.translation);
30 }
31
32 attribute translation<silver:AppExprs> occurs on ASTs;
33
34 aspect production consAST
35 top::ASTs ::= h::AST t::ASTs
36 { top.translation = consAppExpr(h.translation, t.translation); }
37
38 aspect production nilAST
39 top::ASTs ::=
40 { top.translation = nilAppExpr(); }
```

Figure 11: Some of the equations in the SILVER library for computing the translation of `AST` to `silver:Expr`. See silver/metatranslation/Translation.sv

will contain the full names of all known antiquote productions. Extensions, such as the Silver-ableC extension which imports this grammar, can add the names of its antiquote productions to this attribute. This is done using another `aspect` production on line 16 in Figure 10 using the `<-` operator for injecting new elements (names) into this list; these are combined using the list append `++` operator as specified on line 18 in Figure 11. Using this, the name of a production being translated may be looked up, as can be seen in on line 22. If the name is not in the list, the `AST` is translated normally, otherwise it is reified for evaluation. Thus, the `AST` translation code forms a library for use by tree-literal Silver extensions that does not itself have any special handling for particular object-languages.

5.2. Extensions to Antiquotation

The use of antiquoting and object-language concrete syntax has been extended beyond the core implementation described above. Many languages have notions of lists in their grammar, for example a sequence of function arguments; it is possible that one may wish to construct a tree where only a portion of such a list is antiquoted. For example, in

```
ableC_Expr { foo( 3, $Exprs{a}, x ) }
```

the list of expressions `a`, represented by the `Exprs` nonterminal, is to be inserted into the enclosing list of arguments.

The translation of such an antiquote production involves the use of an append function specific to the nonterminal to assemble these lists at run time, for example the above would translate into

```
consExpr(intLiteralExpr(3), appendExpr(a, consExpr(varExpr(name("x")), nilExpr())))
```

where `appendExpr` is a recursive function defined in the ableC host language specification. The implementation of this translation process involves additional collection production attributes and translation cases, similar to those on lines 18 and 22 of Figure 11 (the implementation is not shown here due to its complexity. [7])

5.3. Concrete Object Syntax for Pattern Matching

Similar to the problem of constructing complex syntax trees is the problem of deconstructing them through pattern matching. Silver does pattern matching on trees much the same way that languages such as ML or Haskell do on inductive (algebraic) datatypes. An example of this may be found in the implementation of the ableC-Halide extension [11], inspired by the Halide [20] C++ embedded DSL. The extension allows for iterative computations consisting of multiple nested loops to be expressed separately from optimizing transformations (such as unrolling, tiling or parallelism), allowing for more readable code and greater ease of experimentation with various transformations without fear of introducing errors.

An example use of the extension to perform an optimized matrix multiplication can be found in Figure 12. The computation to be performed is specified as using the syntax derivable from the `Stmt` nonterminal of ableC (lines 4—11), while a series of optimizing transformations on contained `for`-loops (identified by loop variable) are written using a custom DSL (lines 13—21.) For example, splitting a loop converts it into multiple nested loops where all but the outermost run for a constant number of iterations, while unrolling a loop requires duplicating its body a number of times in sequence. Tiling, parallelizing, and vectorizing loops has the expected behavior.

Computing the translation of a `transform` statement requires an intricate series of syntax tree transformations, using a number of attributes. While it would be possible to directly perform these transformations on the `Stmt` nonterminal, this is not the most suitable representation as all loops to be transformed must be of the form

```
for (TYPE i = 0; i < LIMIT; i++) BODY
```

for some variable `i`. A better option is to initially error-check the `Stmts` for well-formedness and convert them to a simpler new `IterStmt` intermediate representation, *e.g.* representing a for loop with the production `forIterStmt` with type

```
IterStmt ::= ty::TypeExpr var::Name limit::Expr body::Stmt
```

```
1   void matmul(unsigned m, unsigned n, unsigned p,
2               float a[m][p], float b[p][n], float c[m][n]) {
3     transform {
4       for (unsigned i = 0; i < m; i++) {
5         for (unsigned j = 0; j < n; j++) {
6           c[i][j] = 0;
7           for (unsigned k = 0; k < p; k++) {
8             c[i][j] += a[i][k] * b[k][j];
9           }
10        }
11      }
12    } by {
13      split i into (unsigned i_outer,
14                    unsigned i_inner : (m - 1) / NUM_THREADS + 1);
15      parallelize i_outer into (NUM_THREADS) threads;
16      tile i_inner, j into (TILE_DIM, TILE_DIM);
17      split k into (unsigned k_outer,
18                    unsigned k_unroll : UNROLL_SIZE,
19                    unsigned k_vector : VECTOR_SIZE);
20      unroll k_unroll;
21      vectorize k_vector;
22    }
23  }
```

Figure 12: An example use of the ableC-Halide extension to implement an optimized matrix multiplication.

```
1   function stmtToIterStmt
2   IterStmt ::= s::Decorated Stmt
3   {
4     return
5       case s of
6       | nullStmt() -> nullIterStmt()
7       | seqStmt(s1, s2) -> seqIterStmt(stmtToIterStmt(s1), stmtToIterStmt(s2))
8       | compoundStmt(s1) -> stmtToIterStmt(s1)
9       | ableC_Stmt { if ($Expr{c}) $Stmt{t} else $Stmt{e} } ->
10        condIterStmt(c, stmtToIterStmt(t), stmtToIterStmt(e))
11      | ableC_Stmt {
12          for ($TypeExpr{t} $Name{i1} = 0;
13               $Name{i2} host::< $Expr{n}; $Name{i3} host::++)
14            $Stmt{b}
15        } when i1.name == i2.name && i1.name == i3.name ->
16        forIterStmt(t, i1, n, stmtToIterStmt(b))
17      | s -> stmtIterStmt(new(s))
18      end;
19  }
```

Figure 13: An example use of concrete object syntax patterns in the ableC-Halide extension, translating host `Stmt` trees into `IterStmts`. See ableC-halide/abstractsyntax/IterStmt.sv

Recognizing and extracting the components of properly-formed for loops is possible with explicit pattern matching, but requires the specification of large and verbose patterns quite similar to the cumbersome and verbose expressions for constructing trees presented previously. An alternative is to use concrete syntax in pattern matching, as shown in Figure 13 on lines lines 9—15. Patterns may be specified using quote productions (such as `ableC_Stmt`) that match on the corresponding term, analogous to quote productions for tree construction.

Pattern anti-quote productions allow for regular SILVER patterns to be written within object-language patterns, in much the same way as regular anti-quote productions allow for the embedding of SILVER expressions within object-language literals. Note that pattern and expression antiquote productions can both syntactically occur within a fragment of object-language code, regardless of whether the fragment is intended to be a pattern or an expression. As the same syntax is used for both types of antiquotation, this results in an ambiguity, resolved through COPPER's features for explicit lexical disambiguation. Alternatively a different syntax for pattern antiquotation could have been chosen, such as `@Expr{...}`

The pattern translation process is quite similar to that used in term construction as seen in Figure 9, except that production, constant, and list `AST` values are translated into SILVER's abstract syntax for patterns that match these values, rather than their construction expressions.

Some object languages have productions in their abstract syntax that further translate or de-sugar down to core features of the language. For example in ABLEC, the parser constructs an abstract syntax tree containing variants of operator productions that may be overloaded for extension types, which are translated (through forwarding) down to different type-specific productions. With concrete object syntax constructs, such forwarding productions can show up in the tree that gets reflected and translated. This isn't an issue for tree construction as these productions will still be translated down properly, but this does present an issue for pattern matching as a pattern for the original production will fail to match an already-translated variant. [8]

This can be remedied by introducing additional concrete syntax that directly constructs the non-forwarding variants of host language productions. This may conflict with the regular concrete syntax for these operators, but conflicts can be disambiguated when building the extended form of SILVER through a mechanism of COPPER known as *transparent prefixes* [21]. For example on line 13 of Figure 13, the (non-forwarding) variants of the `<` and `++` language operators corresponding to host-language types are used by specifying the prefix `host::`. Note that these variants could also be used in tree construction to build trees specific to host language types, however doing so is not particularly useful as the overloaded variants should translate in the same way.

5.4. Additional Applications and the Issue of Bootstrapping

Utilizing additional extensions. In the discussion above, the object-language for which concrete syntax was used was the C host language in the ABLEC system, for the case when new extensions used this to specify how their extension constructs were to be translated down to plain C code. Sometimes, however, language extensions find it useful to build on (that is, translate down to) an extended version of C instead of plain C; both ABLEC and the approach to object-language concrete syntax support this. An example of this can be seen in the extension that adds inductive data types and pattern matching [6] of them as found in languages such as OCaml and Haskell. A further extension (*e.g.* one for unifying terms represented by inductive data types[9]) may wish to construct a translation syntax tree containing match statements from the data type extension, and would like to do so using concrete syntax literals.

There are two steps needed to achieve this. First, the extended SILVER compiler needs to be built such that additional object language extensions, such as the inductive data type extension, are included in the parser, and they are then able to use the extension concrete syntax. Next, anti-quote notations could be used in pattern concrete syntax if the developer of the inductive data type extension provides quote and anti-quote productions for the `Pattern` nonterminal that it has introduced. These are similar in style to those for expressions seen previously in Figure 10.

Other extensible and monolithic object languages. Although this discussion has focused on SILVER-ABLEC, any object-language implemented in SILVER can easily benefit from the ability to write concrete syntax for constructing trees and

---

[7] See silver/metatranslation/Translation.sv in the SILVER grammar.

[8] Note that the opposite is not the case; pattern matching does look through forwarding, falling back to match against forwarded-to trees.

[9] Available at `https://github.com/melt-umn/ableC-unification`, archived at `https://doi.org/10.13020/D6VQ25`.

patterns. Adapting this for another object-language amounts to specifying the concrete and abstract (Figure 10) syntax for quote and antiquote productions for the language, then importing this grammar and the concrete syntax of the object-language into a new extended instantiation of Silver. Additional languages for which object-language concrete syntax extensions have been implemented include Silver itself, to enable the easier development of further extensions to Silver, and the `AST` nonterminal, to provide less verbose syntax for construction of, and pattern matching on `AST` values. We have also added object-language concrete syntax to the Silver implementation of Oberon0, the language of the 2011 LDTA Tool Challenge [22]. One use here was in the implementation of for-loops as syntactic sugar that translates to while-loops[10].

The Oberon0 specification differs from ableC in that it was not written with the intention of being a host language to which various extension would be added. This raises an interesting problem: to build an extended Silver compiler (either Silver-ableC or Silver for Oberon0) a non-extended version of Silver must process at least the concrete syntax specifications of the object language and, thus, these object language specifications cannot contain any uses of the extension to provide concrete syntax for that object language. In extensible languages like ableC this is not an issue as the extensions that require object-language concrete syntax are separate from the host-language defining the syntax; however, this is not necessarily the case in monolithic languages such as Oberon0. The preferred solution for monolithic languages is to separate the implementation into a portion defining the core concrete and abstract syntax of the language, not permitted to use concrete syntax literals, and separate modules containing the implementations of additional features. The extended meta-language compiler can then be built using only the core language, and the result is used to build the complete object-language compiler. Note that in the case of our Oberon0 implementation, the required separation of modules happened to already exist. While this refactoring should be relatively simple for other monolithic language specifications, it is an unfortunate characteristic of our approach that this might be necessary.

Similar considerations must be taken for object-language extensions that wish to use the concrete syntax of other extensions. An extension cannot easily use its own concrete syntax for tree construction, but hierarchies of extensions can exist that each depend on the previous in this way. A downside of this state of affairs is that multiple compilation passes are required to initially build the fully-extended meta-language compiler, each pass including additional extensions in the parser. For example our pre-built distribution of Silver-ableC, containing a number of ableC extensions generally useful in building further extensions, requires 3 passes.

5.5. Parsing and Context-Aware Scanning

Parsing Silver specifications that contain concrete syntax from the object language, such as C, does impose some challenges to traditional parsing and scanning techniques. For example, in scanning the text in Figure 2, the curly brackets on lines 3 and 13 must be recognized as symbols from the Silver specification and those on lines 8 and 10 must be recognized as coming from the ableC specification. This challenge is the same one faced in by extensible languages in which symbols from different extensions, with overlapping regular expressions, must be distinguished. Silver solves this problem by using context-aware scanning [13] as implemented in the underlying Copper parser and scanner generator. This approach uses the state of the generated LR parser to determine the set of terminal symbols that are valid in the current parse state; these are the ones with a shift, reduce, or accept action. The scanner will then only return tokens from this set. While in a Silver-specific parse state, curly brackets are recognized as Silver terminal symbols, but in an ableC parse state they are recognized as terminals from that language. Furthermore, the quote-productions that provide the **`ableC_Expr`** syntax (line 6 in Figure 2) form a extension to Silver and the anti-quote productions that provide the **`$Expr`** and **`$TypeExpr`** syntax inside the C code (lines 7—9) form an extension to ableC. Both of these extensions satisfy the composabilty criteria of the modular determinism analysis [21] and are thus compatible with other independently-developed extensions that also satisfy the criteria. This compatibility means that the composed context-free grammars are LALR(1) and there are no lexical ambiguities in the lexical specification.

5.6. Discussion

In a non-trivial mainstream language such as C, the use of concrete syntax over abstract syntax is a tremendous saver of time and effort. Trees can be specified directly and any syntax errors in the object-language concrete syntax

[10] `https://github.com/melt-umn/Oberon0/blob/v0.1.1/grammars/edu.umn.cs.melt.Oberon0/constructs/controlFlow/abstractSyntax/ForLoop.sv`

```
1  template<a> a min(a x, a y) {          1  int _template_min_int(int x, int y) {
2    if (x < y) return y;                  2    if (x < y) return y;
3    else return x;                  ⟹     3    else return x;
4  }                                       4  }
5  ...                                     5  ...
6  min<int>(i, j)                          6  _template_min_int(i, j)
```

Figure 14: Instantiating an ableC C++-style template extension (left) into plain C code (right) by substitution.

are detected when the language specification is compiled, with error messages pointing to the precise location in the Silver specification[11]. We implemented several extensions to ableC using this new approach, including an embedding of Prolog and a supporting unification framework. These generate quite a bit of plain C code and we may not have attempted these without being able to use of the Silver-ableC extension to Silver. We did modify the implementation [12] to count the number of characters that are saved by writing in the object-language concrete syntax (Figure 2) instead of the abstract (Figure 3.) This was done by pretty-printing the abstract syntax generated as the forward of all quote productions, and comparing it with the size of the quoted syntax literals. For all extensions that use Silver-ableC these savings are over 775K characters (or almost 18K lines of code based on pretty-printing the generated code with a maximum line length of 80). This is about 40% of the would-be code base size of 1.92M characters. Note that some of this saving came from writing new extensions that used this feature from the beginning, not from removing and replacing specifications with the concrete syntax extension.

The extension to Silver for using Oberon0 concrete syntax in its specification was less dramatic since complex syntax trees are only built directly in a few places, such as in de-sugaring for-loops into while-loops. While the use of this feature saved a few lines (11, in fact) in the Oberon0 specification, the specification of the extension used 224 lines, resulting in a net increase in the size of the overall specification. Thus the overhead of writing such an extension to Silver must be balanced with how much it will be used.

6. Reflection for Term Rewriting

A common problem with processing complex tree-structured data is to update or substitute the value of a particular sub-term, without introducing large amounts of boilerplate. This sort of problem may arise in a compiler when implementing a feature such as C++ templates, in which we wish to replace all occurrences of a name with the term for a particular type or expression. An example of this transformation (shown in Figure 14) comes from ableC in which a C++-style template extension (left) is instantiated to plain C code (right). This sort of transformation may be implemented using higher-order attributes, however doing so would require a large amount of boilerplate specification, specifically, writing an attribute equation for each production in the grammar. More efficient approaches, based on type classes, have also been proposed [4], but in a system with a less sophisticated type system, such as Silver, substitution may also be expressed concisely through the use of reflection. In this section we develop a general strategy-based term-rewriting extension to Silver based on reflection. This is used here to solve the substitution problem in ableC and to implement a λ-calculus evaluator based on a previous solution encoded in Stratego [10].

It is also possible to directly implement substitution as a library using reflection. In the earlier conference version of this paper [9] we presented an approach that works by representing substitutions as a list of functions (of type `Maybe<AST> ::= AST`), each encoding an attempted transformation on a sub-term. However this approach has several drawbacks:

- Complexity: Specifying a new kind of substitution requires either matching directly on a complex `AST` to determine whether the term is to be substituted, or inefficiently attempting to reify and match on every sub-term.

[11] The source location of a tree in Silver is represented as an *annotation*. These are extra pieces of information attached to syntax tree nodes, but are omitted here for clarity as they are handled essentially the same as regular children. Thus location information is preserved in a reflected `AST`.

[12] Available at `https://github.com/melt-umn/silver/tree/concrete-object-syntax-stats`.

```
1  function substTypeExprInDcl
2  Decl ::= n::String ty::TypeExpr d::Decl
3  {
4    local substType::Strategy =
5      allTopDown(
6        rule on TypeExpr of
7        | typedefTypeExpr(_, n1) when n1.name == n -> ty
8        end);
9    return
10     case rewriteWith(substType, d) of
11     | just(d1) -> d1
12     | nothing() -> error("This strategy should always succeed")
13     end;
14 }
```

Figure 15: A hypothetical function to replace a named type with a type expression within a declaration. The actual implementation of templates is more complex; see ableC-templating/abstractsyntax

- Lack of generality: Some kinds of transformations, such as capture-avoiding substitution, require more detailed control over the traversal order.
- Lack of introspection: Functions are "opaque", thus we can't print out substitutions for debugging, serialize them to a file, etc.

Term rewriting [23] is a powerful formalism for generic program transformations that avoids these problems. The Stratego [24] rewriting system popularized the notion of *rewriting strategies*, a domain-specific language of expressions for specifying term transformations. Strategies are built from atomic rewrite rules and a number of combinators for controlling the traversal order, and when applied to a term can either succeed with a new term or fail with no value.

Using reflection, we can add features for strategic term rewriting to Silver, taking heavy inspiration from Stratego. This rewriting system can then be used to solve the substitution problem much more elegantly. Figure 15 shows a (simplified) example addressing the template substitution problem from Figure 14, in which we wish to replace names (parsed as references to `typedef` types) with type expressions. Here we define a strategy (lines 4—8) that performs a top-down traversal and tries to replace each matching type expression with a new type expression (`ty`). This strategy is applied to a tree (line 10) resulting in a `Maybe` value of just a successful result or nothing in case of failure, however the `allTopDown` strategy is defined in such a way that its application will never fail. The actual implementation of substitution used by the template extension is more complex, constructing strategies separately from where they are applied, and substituting multiple template parameters in the same rewriting pass.

The term rewriting feature of Silver is divided into two components. First we will look at how reflection may be used to implement a Silver library for representing rewriting strategies and applying them to terms (Section 6.1.) We will then present a separate set of language extensions to Silver that provide more convenient (and type-safe) syntax for specifying and applying strategies (Section 6.2), also examining uses of reflection in the extensions' implementation. Finally we will explore some more advanced features of rewriting, in conjunction with larger example applications of the rewriting system (Sections 6.3 and 6.4.)

6.1. *Design of the Rewriting Library*

In examining the design of the rewriting library, we will use as an example a bottom-up replacement of all occurrences of $x + 0$ with x for any sub-expression x. This transformation is specified by the `elimPlusZero` strategy (to be discussed later); we will first analyze the application of this strategy to an `Expr` term, as shown in Figure 16. Rewriting is done on terms represented as `AST` values; the term `e` being rewritten is first reflected to an `AST` (line 4) and the final `AST` result is reified (line 9.) Applying a strategy to a term may either succeed with a new resulting term or

```
1  global elimPlusZero::Strategy = ...;     -- see Figure 17
2  function elimPlusZeroInExpr
3  Expr ::= e::Expr
4  { local a :: AST = reflect(e);
5    return
6      case decorate elimPlusZero with {term = a;}.result of
7      | nothing() -> error("This strategy should always succeed")
8      | just(aRes) ->
9        case reify(aRes) of
10       | left(msg) -> error("Rewriting produced ill-sorted AST: " ++ msg)
11       | right(res) -> res
12       end
13     end;
14 }
```

Figure 16: The application of strategy `elimPlusZero` to perform a bottom-up rewrite of $x + 0 \to x$ on ABLEC `Expr` syntax trees and its definition.

fail with no value. However this example strategy should never fail when applied (line 7), as with the earlier example of type substitution (line 12 of Figure 15.)

Strategies such as `elimPlusZero`, defined in Figure 17, are represented by the `Strategy` nonterminal. This is defined on line 3 of Figure 18, which contains part of the term rewriting library. Two attributes, `term` and `result` on lines 4—5, on this nonterminal are used to apply a strategy to a term. The `term` attribute passes the AST to be rewritten down the `Strategy` tree, while `result` passes back up the tree either just a successfully-rewritten `AST` or nothing in case of a failure. Thus to apply a strategy to an `AST` value, the `Strategy` is decorated with the `AST` as the inherited value for `term`, and the `result` is accessed on the resulting decorated `Strategy` tree (line 6 of Figure 16.)

```
1  global elimPlusZero::Strategy =
2    sequence(
3      all(elimPlusZero),
4      choice(
5        -- addExpr(a, intExpr(0)) -> a
6        rewriteRule(
7          prodCallASTPattern(
8            "ableC:addExpr",
9            consASTPattern(
10             varASTPattern("a"),
11             consASTPattern(
12               prodCallASTPattern(
13                 "ableC:intExpr",
14                 consASTPattern(
15                   intASTPattern(0),
16                   nilASTPattern())),
17               nilASTPattern()))),
18          varASTExpr("a")),
19       id()));
```

Figure 17: The definition of a strategy to perform a bottom-up rewrite of $x + 0 \to x$ on ABLEC syntax trees.

As with STRATEGO, basic strategies are built from a number of combinator productions. An example of strategy construction using them is shown in Figure 17 for the definition of the `elimPlusZero` strategy. These are listed in Figure 18 and discussed below. The `id()` strategy (line 7 of Figure 18) always succeeds and returns as its result (in the synthesized `result` attribute) the input term passed in as the inherited attribute `term`. The `fail()` (line 11) strategy always fails by returning a `nothing()` value, ignoring the input term. The `choice` (line 15) strategy attempt to rewrite the term with the first strategy `s1` and if this fails,[13] uses the result of trying with the second strategy `s2`. The `sequence` (line 27) strategy attempts the rewriting with two strategies in sequence. If the first succeeds, then the result on `s2` is returned. The input term to `s2` is only accessed if `s1` succeeds. Thus the (standard library) `fromJust` attribute that unwraps the value under a `just` in a

[13] Our version of choice is deterministic (it does not backtrack and try the try operand if a failure occurs later on after the left operand originally succeeded.) STRATEGO offers both deterministic and nondeterministic choice strategy constructors, however nondeterministic choice is not recommended for use in practice and is mainly used in the translations of other strategy primitives.

Maybe value (and raises and error in the case of a nothing value) will always succeed here. Note that this idiom is commonly used and is seen in additional examples in the paper.

Basic strategies. The most fundamental type of strategy is the rewrite rule, which matches a pattern against the current term and either succeeds and replaces the term with a new term, or fails and does nothing. Rule strategies are built by the rewriteRule constructor (line 39), parameterized by a pattern and a result expression that should be evaluated to replace the term when it matches the pattern. Note that the strategies containing these patterns and expressions are built at runtime, and must operate on the generic AST representation of values. So instead of compiling these patterns and expressions like the ones built into Silver, we have effectively built an interpreter that operates using AST values at runtime.

While we could represent the components of rewriteRule by Silver's own Pattern and Expr nonterminals, for a few idiosyncratic reasons[14] it is simpler to define separate run-time representations. These representations are provided by the ASTPattern and ASTExpr nonterminals, which have productions corresponding to every basic type of pattern and expression in Silver. In Figure 17, the rule's pattern can be seen on lines 7—17, and the expression on line 18. These correspond to matching the term against the Silver pattern addExpr(a, intExpr(0)) and, if successful, replacing the term with the value of a.

A definition of the ASTPattern nonterminal and some of it's productions are given in Figure 19. When evaluating a rewriteRule strategy, the pattern must first be unified with the current term. This is done by passing the term's AST value down the ASTPattern tree using the inherited matchWith attribute. If the pattern does not match, the substitution synthesized attribute has the value nothing(). If it does match then a substitution (wrapped in a just) matching names to terms, represented as a list of String and AST pairs is computed. The prodCallASTPattern production checks that it is being matched against a nonterminalAST term with the correct production name (line 13), before recursively supplying its children with the appropriate child AST values of the given AST (line 9.) Literal patterns such as integerASTPattern must simply check for equality with the AST given by matchWith (line 23), while variable and wildcard patterns successfully match against anything. When matched against, varASTPattern will construct a substitution consisting of the variable name and the given AST (line 29.)

When the match succeeds the rule's expression must then be evaluated using the substitution; this is done with the inherited substitutionEnv and synthesized value attributes on the ASTExpr nonterminal, as shown in Figure 20. On prodCallASTExpr a nonterminalAST is built from the recursive results of evaluating the children in the same substitution environment (line 11.) Evaluating a varASTExpr requires looking up a name from the environment (line 24), and operators such as addition require matching on the values of the children (lines 36—40.) Note that a fatal run-time error is raised if an unbound variable or type error is encountered; these are considered bugs in the definition of the strategy, rather than cases that might result in a rewriting failure such as rule failing to match.

Generic traversal strategies. Recursive transformations across a tree may be accomplished using generic traversal strategies such as all, some and one. For example, in Figure 17, the all strategy is used to apply elimPlusZero to all children of the current term, succeeding only in case of success for all children, on line 3; note that this strategy is constructed recursively as an infinite term by taking advantage of Silver's laziness. Similarly, some succeeds if at least one child succeeds, and one applies its argument to one child at a time and succeeds after the first child's success. This is done in a deterministic fashion - one does not backtrack and try rewriting additional children if a failure occurs later on.

The implementations of these strategies are all quite similar, but the implementation of all can be seen in Figure 21. Here instead of threading the term through the strategy with attributes, we pass the strategy into the term with the givenStrategy inherited attribute. This is because traversal strategies must behave differently depending on the structure of the term, as opposed to strategies such as choice and sequence that only affect the control flow and do not inspect the current term.

Most AST productions, such as integerAST have no AST children; for these productions successfully applying a strategy to all children (of which there are none) amounts to doing nothing. This default behavior is specified on

[14] Some of the Silver abstract syntax uses forwarding to de-sugar constructs to a core syntax, and this process requires providing inherited attributes to that syntax. Additionally terms in Silver's syntax trees are annotated with information about the locations from which they were parsed in the source file, which is not required at runtime. This makes the chosen approach easier to manage.

```
grammar silver:rewrite;

nonterminal Strategy with term, result;
inherited attribute term::AST;
synthesized attribute result::Maybe<AST>;

abstract production id
top::Strategy ::=
{ top.result = just(top.term); }

abstract production fail
top::Strategy ::=
{ top.result = nothing(); }

abstract production choice
top::Strategy ::= s1::Strategy s2::Strategy
{
  s1.term = top.term;
  s2.term = top.term;
  top.result =
    case s1.result of
    | just(a) -> just(a)
    | nothing() -> s2.result
    end;
}

abstract production sequence
top::Strategy ::= s1::Strategy s2::Strategy
{
  s1.term = top.term;
  s2.term = s1.result.fromJust;
  top.result =
    case s1.result of
    | just(_) -> s2.result
    | nothing() -> nothing()
    end;
}

abstract production rewriteRule
top::Strategy ::= pattern::ASTPattern result::ASTExpr
{
  pattern.matchWith = top.term;
  result.substitutionEnv = pattern.substitution.fromJust;
  top.result =
    case pattern.substitution of
    | just(_) -> just(result.value)
    | nothing() -> nothing()
    end;
}
```

Figure 18: The implementations of some of the basic strategy constructors. See silver/rewrite/Strategy.sv

```
1  grammar silver:rewrite;
2  nonterminal ASTPattern with matchWith<AST>, substitution;
3  inherited attribute matchWith<a>::a;
4  synthesized attribute substitution::Maybe<[Pair<String AST>]>;
5
6  abstract production prodCallASTPattern
7  top::ASTPattern ::= prodName::String children::ASTPatterns
8  {
9    children.matchWith = -- Only demanded when we have a nonterminalAST
10     case top.matchWith of nonterminalAST(_, c) -> c end;
11   top.substitution =
12     case top.matchWith of
13     | nonterminalAST(otherProdName, _) when prodName == otherProdName ->
14       just(childrenSubstitution);
15     | _ -> nothing()
16     end;
17 }
18 abstract production integerASTPattern
19 top::ASTPattern ::= i::Integer
20 {
21   top.substitution =
22     case top.matchWith of
23     | integerAST(i1) when i == i1 -> just([])
24     | _ -> nothing()
25     end;
26 }
27 abstract production varASTPattern
28 top::ASTPattern ::= n::String
29 { top.substitution = just([pair(n, top.matchWith)]); }
30
31 abstract production wildASTPattern
32 top::ASTPattern ::=
33 { top.substitution = just([]); }
```

Figure 19: Some of the productions and attribute equations for `ASTPattern`. See silver/rewrite/ASTPattern.sv

line 17 using a *aspect default production*, indicating that every production on `AST` lacking an equation for `allResult` should define its value to be just the `AST` value unchanged. On `nonterminalAST`, we thread the strategy down to the children and attempt to apply it to each child (lines 30—45.) If all succeed, we return just the rewritten children wrapped in `nonterminalAST` with the same production, or otherwise fail with nothing (lines 24—27.)

6.2. *Extensions to* Silver *for Rewriting*

As seen above, using the term rewriting system through the Silver library API is verbose and provides no assurances of type-safety with respect to the syntax of the object language. Since Silver is itself an extensible language we have extended it with new syntax and static semantics to address these concerns.

6.2.1. *Infix operators and their use in library strategy productions*

One such useful extension provides new infix operators to make strategy construction easier. The `s1 <+ s2` infix operator is an alternative notation for `choice(s1, s2)`, the same as in Stratego. Similarly, the `s1 <* s2` infix operator is an alternative notation for `sequence(s1, s2)`. This differs from the `;` sequence operator syntax provided

```
1  grammar silver:rewrite;
2
3  nonterminal ASTExpr with substitutionEnv, value;
4  inherited attribute substitutionEnv::[Pair<String AST>];
5  synthesized attribute value::AST;
6
7  abstract production prodCallASTExpr
8  top::ASTExpr ::= prodName::String children::ASTExprs
9  {
10   children.substitutionEnv = top.substitutionEnv;
11   top.value = nonterminalAST(prodName, children.values);
12 }
13
14 abstract production integerASTExpr
15 top::ASTExpr ::= i::Integer
16 {
17   top.value = integerAST(i);
18 }
19
20 abstract production varASTExpr
21 top::ASTExpr ::= n::String
22 {
23   top.value =
24     case lookupBy(stringEq, n, top.substitutionEnv) of
25     | just(a) -> a
26     | nothing() -> error("Unbound variable " ++ n),
27     end;
28 }
29
30 abstract production plusASTExpr
31 top::ASTExpr ::= a::ASTExpr b::ASTExpr
32 {
33   a.substitutionEnv = top.substitutionEnv;
34   b.substitutionEnv = top.substitutionEnv;
35   top.value =
36     case a.value, b.value of
37     | integerAST(x), integerAST(y) -> integerAST(x + y)
38     | floatAST(x), floatAST(y) -> floatAST(x + y)
39     | _, _ -> error("Invalid values")
40     end;
41 }
42
43 -- Represents other kinds of literals, e.g. functions
44 abstract production anyASTExpr
45 top::ASTExpr ::= x::a
46 {
47   top.value = reflect(x);
48 }
```

Figure 20: Some of the productions and attribute equations for `ASTExpr`. See silver/rewrite/ASTExpr.sv

```
grammar silver:rewrite;

abstract production all
top::Strategy ::= s::Strategy
{
  local term::AST = top.term;
  term.givenStrategy = s;
  top.result = term.allResult;
}

inherited attribute givenStrategy::Strategy occurs on AST, ASTs;
synthesized attribute allResult<a>::Maybe<a>;
attribute allResult<AST> occurs on AST;

aspect default production
top::AST ::=
{ top.allResult = just(top); }

aspect production nonterminalAST
top::AST ::= prodName::String children::ASTs
{
  children.givenStrategy = top.givenStrategy;
  top.allResult =
    case children.allResult of
    | just(a) -> just(nonterminalAST(prodName, a))
    | nothing() -> nothing()
    end;
}

attribute allResult<ASTs> occurs on ASTs;
aspect production consAST
top::ASTs ::= h::AST t::ASTs
{
  t.givenStrategy = top.givenStrategy;
  top.allResult =
    case decorate top.givenStrategy with { term = h; }.result,
         t.allResult of
    | just(hResult), just(tResult) -> consAST(hResult, tResult)
    | _, _ -> nothing()
    end;
}

aspect production nilAST
top::ASTs ::=
{ top.allResult = just(top); }
```

Figure 21: A portion of the implementation of the `all` traversal strategy. Equations for `listAST` are hidden for clarity.

by STRATEGO, as `;` in SILVER is used to terminate expressions and would be syntactically ambiguous; instead `<*` is the sequence operator used in the strategic rewriting portion of the KIAMA [1] language processing system. Example uses of these operators can be found in the rewriting library, where a number of commonly-useful patterns are included as `Strategy` productions that translate to their implementations:

- The `try :: (Strategy ::= s::Strategy)` strategy translates to `s <+ id()`; it simply tries applying its parameter strategy, but always succeeds.
- `repeat :: (Strategy ::= s::Strategy)` translates to `try(s <* repeat(s))`; this strategy tries applying its argument repeatedly until the first failure, and then succeeds.
- `bottomUp :: (Strategy ::= s::Strategy)` translates to `all(bottomUp(s)) <* s`; it applies a strategy once to every sub-term of the term, beginning from the leaf members of the term, and fails if any sub-term application fails. A similar `topDown` strategy is also provided.
- `allTopDown :: (Strategy ::= s::Strategy)` translates to `s <+ all(allTopDown(s))`; it applies a strategy once to every sub-term of the term, beginning from the top term, but stopping in a sub-term after the first success. This strategy never fails. A similar `allBottomUp` strategy is also provided.
- `innermost :: (Strategy ::= s::Strategy)` translates to `bottomUp(try(s <* innermost(s)))`; it repeatedly applies a strategy to the innermost, leftmost expression in a term, only moving up the tree once all sub-terms are fully reduced.
- The `rec :: (Strategy ::= ctr::(Strategy ::= Strategy))` strategy is useful in constructing recursive strategies in-line. When given a function `ctr` (usually defined as a lambda), it forwards to the result of applying `ctr` with itself as the parameter. Recursive strategies can also be defined directly through the use of laziness, as seen in some of the above utility strategies such as `repeat`, or in Figure 17 on line 3.

Using these library strategies, the `elimPlusZero` strategy from Figure 17 could be defined as

```
global elimPlusZero::Strategy = bottomUp(try(rewriteRule(..., ...)));
```

instead of directly specifying a recursive traversal.

We also provide convenience syntax **`rewriteWith`**`(`*`STRATEGY`*`, `*`TERM`*`)` as a shorthand for applying a strategy to a term (lines 6—13 of Figure 16.) This syntax was also previously seen on line 10 of Figure 15 to apply the `substType` strategy to a `TypeExpr` term.

6.2.2. Specifying rewrite rules

While it is possible to directly construct rule strategies using `rewriteRule`, doing so has a number of drawbacks: such strategies are verbose and hard to read (as seen in the definition of `elimPlusZero` in Figure 17) as well as lacking any type checking. Type errors can arise in a number of ways when creating `ASTExprs` and `ASTPatterns` directly - the types of the rule and pattern in a rule can differ, a production can be called with the wrong number or type of arguments, an undefined variable may be referenced, etc.

These problems are addressed by an additional language extension to SILVER, providing better syntax for specifying rewrite rules. This extension reuses the syntax and semantics of ordinary pattern matching in SILVER for parsing and type checking, but transforms the component patterns and expressions into `ASTPattern` and `ASTExpr` trees for evaluation. For example, the `rewriteRule` on lines 6—18 of Figure 17 can be replaced by

```
rule on Expr of addExpr(a, intExpr(0)) -> a end
```

While this instance of the **`rule`** construct provides only one rewrite rule for expressions it does allow a sequence of rewrite rules to be specified over a nonterminal type, just like the clauses in a typical SILVER `case` expression for matching patterns against a value. More generally, the expression

```
rule on TYPE of MATCH_RULES end
```

```
1  grammar silver:extension:rewriting;
2  imports silver:metatranslation;
3  concrete production ruleExpr
4  top::Expr ::= 'rule' 'on' ty::TypeExpr 'of' '|' ml::MatchRuleList 'end'
5  { forwards to reflect(ml.transform).translation; }
6
7  synthesized attribute transform<a>::a;
8
9  attribute transform<Strategy> occurs on MatchRuleList, MatchRule;
10 aspect production oneMatchRule
11 top::MatchRuleList ::= m::MatchRule
12 { top.transform = m.transform; }
13
14 aspect production consMatchRule
15 top::MatchRuleList ::= h::MatchRule '|' t::MatchRuleList
16 { top.transform = h.transform <+ t.transform; }
17
18 aspect production matchRule
19 top::MatchRule ::= pt::Pattern '->' e::Expr
20 { top.transform = rewriteRule(pt.transform, e.transform); }
21
22 attribute transform<ASTPattern> occurs on Pattern;
23 aspect production prodPattern
24 top::Pattern ::= prod::QualifiedName '(' ps::Patterns ')'
25 { top.transform = prodCallASTPattern(prod.fullName, ps.transform); }
26
27 aspect production intPattern
28 top::Pattern ::= num::Int_t
29 { top.transform = integerASTPattern(toInteger(num.lexeme)); }
30
31 aspect production varPattern
32 top::Pattern ::= n::Name
33 { top.transform = varASTPattern(n.name); }
34
35 attribute transform<ASTExpr> occurs on Expr;
36 aspect production productionCall
37 top::Expr ::= prod::QualifiedName es::Decorated AppExprs
38 { top.transform = prodCallASTExpr(prod.fullName, es.transform); }
39
40 aspect production intConst
41 top::Expr ::= i::Int_t
42 { top.transform = integerASTExpr(toInteger(i.lexeme)); }
43
44 aspect production lexicalLocalReference
45 top::Expr ::= n::Name
46 { top.transform = varASTExpr(n.name); }
47
48 aspect production plus
49 top::Expr ::= e1::Expr '+' e2::Expr
50 { top.transform = plusASTExpr(e1.transform, e2.transform); }
```

Figure 22: A (simplified) portion of the implementation for the **`rule`** strategy constructor. All code related to type checking is omitted for clarity. See silver/extension/rewriting

constructs a strategy that, when applied to a term of the specified type, will perform the first rewrite corresponding to a succeeding match rule.

A simplified implementation of `rule` is shown in Figure 22. The production for a rule, `ruleExpr`, specifies the type (`ty`) of term to which the collection of rewrite rules (`ml`, of type `MatchRuleList`) will be applied. The nonterminal `MatchRuleList` is used in the core Silver pattern matching construct and derives a list (via `oneMatchRule` and `consMatchRule` productions) of the clause form (`matchRule`) that is reused here for term rewriting. By the concrete syntax of pattern matching and expressions any valid Silver pattern or expression can be written in a `rule` body. Each match rule then becomes a `rewriteRule` strategy (line 20). The rules in the sequence are combined, in the `transform` attribute on the aspect productions `oneMatchRule` and `consMatchRule`, into a single strategy using the choice combinator (line 16.) Patterns and expressions are also transformed into the corresponding `ASTPatterns` and `ASTExprs`. For each production that would ordinarily be compiled to operate or match on an ordinary Silver value, we instead transform it to the equivalent `ASTPattern` or `ASTExpr` that will be interpreted on an `AST` value at run time. For example a Silver `prodPattern` becomes a `prodCallASTPattern` (line 25), and a Silver `plus` expression becomes a `plusASTExpr` (line 50.)

The top-level `ruleExpr` production (line 3) must forward to a Silver `Expr` that constructs the desired `Strategy` term value when the program is run, however the translation code in Figure 22 constructs this value at compile time. We can perform the required meta-translation using reflection by reusing the machinery developed for translating embedded object-language literals into meta-language code, from Section 5. This is done on line 5 by reflecting the `Strategy` constructed by `ml.transform` into an `AST`, and computing the `translation` attribute from the meta-translation library on the result to obtain the desired Silver `Expr` term.

Note that we could have alternatively had `transform` directly build the Silver `Expr` needed to construct the desired `Strategys`, `ASTPatterns` and `ASTExprs`. However this implementation would be less maintainable as type errors in the translation would show up when using Silver to build programs including `rule` expressions, rather than immediately when building the Silver compiler.

Lambda calculus example. A more interesting example involving the use of `rule` is given in Figure 23. This is an implementation of the λ-calculus[15] based on an implementation of the same in Stratego [10], with a rule for let-elimination (line 38) taken from a similar example in Kiama [16]. Lambda abstraction (line 10), application (line 14), and variable (line 6) terms are represented here as productions of a Silver nonterminal `Term`. The synthesized attribute `freevars` collects the free variables in a `Term`.

We can use strategies to express evaluation via β-reduction of λ-terms (line 24.) This involves performing a capture-avoiding substitution over a term. Performing this sort of operation with rewriting presents a challenge, as unlike the ableC template extension, we cannot just perform a single top-down pass that replaces all instances of the variable.

Instead, following the Stratego specification, we can introduce an additional kind of term, `letT` (line 18, named as such to avoid conflicting with the `let` keyword in Silver), that only exists during the rewriting process. Let terms are introduced by β-reduction (line 24) of the application of an abstraction. Additional rules (line 30) distribute the introduced lets, pushing them down until reaching a variable or an abstraction that shadows the bound name. This can be seen, for example, on line 34 where the `letT` term is pushed down into a `app` term.

This example utilizes several additional Silver features that must be translated into `Strategy`, `ASTPattern`, and `ASTExpr` values. Guarded `when` patterns evaluate an expression to determine whether the pattern matches. The translation of these involves the `require` strategy, shown in Figure 24. Like `rewriteRule`, `require` matches the term against a pattern and (if the match succeeds) evaluates an expression in the resulting substitution environment. The result of this expression should be a Boolean; if true the strategy succeeds and does not modify the term, otherwise the strategy fails. The `require` strategy can then be used in the translation of `when`, by producing a sequence of a `require` and a `rewriteRule`, both with the same pattern - thus the match is performed twice. For example,

```
rule on Integer of a when a != 42 -> a + 1 end
```

[15] Version 0.1.1, available at `http://melt.cs.umn.edu/` and `https://github.com/melt-umn/lambda-calculus`, archived at `https://doi.org/10.13020/xcfv-5k29`.

[16] `https://github.com/inkytonik/kiama/blob/master/extras/src/test/scala/org/bitbucket/inkytonik/kiama/example/lambda/Lambda.scala`

```
1  grammar lambdacalc;
2
3  nonterminal Term with freeVars;
4  synthesized attribute freeVars::[String];
5
6  abstract production var
7  top::Term ::= id::String
8  { top.freeVars = [id]; }
9
10 abstract production abs
11 top::Term ::= id::String body::Term
12 { top.freeVars = removeBy(stringEq, id, body.freeVars); }
13
14 abstract production app
15 top::Term ::= t1::Term t2::Term
16 { top.freeVars = unionBy(stringEq, t1.freeVars, t2.freeVars); }
17
18 abstract production letT -- Name avoids conflict with Silver let keyword
19 top::Term ::= id::String t::Term body::Term
20 { top.freeVars =
21     unionBy(stringEq, t.freeVars, removeBy(stringEq, id, body.freeVars)); }
22
23 -- Beta reduction with explicit substitution
24 global beta::Strategy =
25   rule on Term of
26   | app(abs(x, e1), e2) -> letT(x, e2, e1)
27   end;
28
29 -- Let distribution
30 global letDist::Strategy =
31   rule on Term of
32   | letT(x, e, var(y)) when x == y -> e
33   | letT(x, e, var(y)) -> var(y)
34   | letT(x, e0, app(e1, e2)) -> app(letT(x, e0, e1), letT(x, e0, e2))
35   | letT(x, e1, abs(y, e2)) ->
36     let z::String = freshVar()
37     in abs(z, letT(x, e1, letT(y, var(z), e2))) end
38   | letT(x, _, e) when !containsBy(stringEq, x, e.freeVars) -> e
39   end;
40
41 -- Full eager evaluation, including reduction inside lambdas
42 global evalInnermost::Strategy = innermost(beta <+ letDist);
```

Figure 23: An implementation of the λ-calculus based on rewriting. See lambdacalc/term_rewriting/Eval.sv

```
1  grammar silver:rewrite;
2  abstract production require
3  top::Strategy ::= pattern::ASTPattern cond::ASTExpr
4  {
5    pattern.matchWith = top.term;
6    cond.substitutionEnv = pattern.substitution.fromJust;
7    top.result =
8      case pattern.substitution of
9      | just(_) ->
10       case cond.value of
11       | booleanAST(b) -> if b then just(top.term) else nothing()
12       | _ -> error("require condition should return a boolean")
13       end
14     | nothing() -> nothing()
15     end;
16 }
```

Figure 24: The `require` strategy constructor.

translates as

```
sequence(
  require(varPattern("a"), eqeqASTExpr(varASTExpr("a"), integerASTExpr(42))),
  rewriteRule(varPattern("a"), plusASTExpr(varASTExpr("a"), integerASTExpr(1))))
```

Both `require` and `rewriteRule` match the same pattern (`varPattern("a")`). This inefficiency could potentially be removed by introducing a second form of `rewriteRule` that includes this guard.

6.2.3. Dealing with nontrivial expressions in rules

Occasionally expressions may occur on the right side of a rule that cannot be directly translated as `ASTExpr`s. For example on line 38 of Figure 23, we call a function (`containsBy`) and access an attribute (`freeVars`.)[17] This presents two problems:

1. The abstract syntax of `ASTExpr` can't represent references to functions or other values that are not either productions or variables provided by the corresponding pattern - in Figure 22 only references to pattern variables (line 44) can be translated as a `varASTExpr`, while an alternative solution is needed to translate other sorts of references. A similar problem exists for λ-expressions occurring on the right side of a rule.
2. It is impractical to interpret complex operations (such as decoration and attribute access) on `AST` values, as this is a generic representation of undecorated terms that does not involve attributes. Thus we require some sort of escape hatch back to actual Silver values and expressions when evaluating the right side of a rule.

To solve the first problem, the Silver `Expr` into which the **`rule`** is translated must contain, besides calls to `Strategy` / `ASTPattern` / `ASTExpr` constructors, some direct references to variables and functions - things that cannot be directly represented within the `Strategy` term value that we construct at compile time in Figure 22.

A solution involves leveraging the antiquotation feature of the meta-translation mechanism from Section 5. We add a new production `antiquoteASTExpr` (Figure 25) wrapping a Silver `Expr` that should be directly incorporated into the final translation. A reference to a function can then transform into, for example,

[17] Although a rule may access attributes on a tree during rewriting, rewriting is still only done on undecorated terms and not on trees that have been decorated with attributes. In the case of such an attribute access, all required inherited attributes must be supplied by the rule expression (in the case of `freeVars` there are none) and the decorated tree is immediately discarded after the evaluation of the rule.

```
1 grammar silver:extension:rewriting;
2 abstract production antiquoteASTExpr
3 top::ASTExpr ::= e::Expr
4 { forwards to error("no forwarding"); }
5
6 aspect production nonterminalAST
7 top::AST ::= _ _ _
8 { directAntiquoteProductions <-
9     ["silver:extension:rewriting:antiquoteASTExpr"]; }
```

Figure 25: Antiquotation for `ASTExpr`.

```
1 antiquoteASTExpr(
2   silver:applyExpr(
3     silver:varExpr("silver:extension:rewriting:anyASTExpr"),
4     silver:consExpr(
5       silver:varExpr("core:containsBy"),
6       silver:nilASTExpr())))
```

which will become `anyASTExpr(containsBy)` in the final meta-translation. When interpreted, such `ASTExpr`s will become `anyAST`s wrapping function values.

The second problem, of evaluating expressions that cannot be easily evaluated on `AST`, can be dealt with by wrapping the ordinary Silver expressions for these operations inside λ-expressions (introduced using `antiquoteASTExpr`) that are immediately applied.

This leaves us with the task of interpreting `ASTExpr`s expressing the application of arbitrary functions. We must apply a function, wrapped in `anyAST`, to a list of arguments also represented as `AST` values. This isn't directly possible using **`reflect`** and **`reify`**, because we must check that the arguments match the type of the function, and we don't statically know the function type.

The solution involves a new primative function

```
applyAST :: (Either<String AST> ::= fn::AST args::[Maybe<AST>])
```

that (partially) applies an `AST` function value (`fn`) to a list of possibly present arguments (`args`), resulting in either a type error message or `AST` value. This function is part of the reflection library, as it requires a dynamic type check between the function and its arguments that cannot be performed in ordinary Silver code (**`reify`** only compares the type of an `AST` with the statically-inferred type of an expression.) The **`applyAST`** function does use **`reify`** internally, with the associated slight performance penalty. However in practice most rewrite rules don't require the use of **`applyAST`**. The **`applyAST`** function is also of more general utility, as it allows us to effectively "downcast" an `AST` value to a type that is only determinable at runtime, as opposed to **`reify`** which requires a specific result type to be inferred at compile-time.

6.2.4. Dealing with polymorphism in patterns

The matching that we perform on `AST` values only considers the structure of the term and not its type. For example, consider the following (admittedly strange) rule:

```
rule on Expr of a -> addExpr(a, a) end
```

The left side of the pattern will match even if the term being rewritten is not an `Expr` (*e.g.* a `Stmt`). This means that we would erroneously replace a `Expr` with a `Stmt` and the final `AST` would fail to reify, which is clearly not the desired behavior. Similar issues can also arise in cases involving generalized algebraic data type productions in abstract syntax (which Silver supports.)

More generally, this is a problem when the structure of a pattern does not constrain the type of a pattern variable (i.e. the pattern is polymorphic, matching more than one type of term.) However the rule is still specified for a

```
1  grammar silver:rewrite;
2  abstract production requireType
3  top::Strategy ::= fn::ASTExpr
4  {
5    top.result =
6      case applyAST(fn.value, [just(top.term)]) of
7      | left(msg) -> nothing()
8      | right(_) -> just(top.term)
9      end;
10 }
```

Figure 26: The `requireType` strategy constructor.

particular type (`Expr` in this example), against which the pattern and expression have been statically type checked. In most cases, the outermost production in the pattern will fail to match against any type of term but the specified one, however in these uncommon cases we must take a different approach to ensure that the rule fails for other types of terms.

Our approach introduces another strategy `requireType` (Figure 26) for the translation of such rules, used to perform a run-time type check of the current term (rules whose type is precisely constrained by the structure of the left hand side pattern can skip this check for better performance.) This strategy is constructed from a λ-expression of one argument represented as `ASTExpr` (the body does not matter, it can return `unit()`.) When applied `requireType` attempts to apply the function using **`applyAST`**, succeeding only if the call does not result in a type error.

Type safety of rewrite rules. Note that performing these added checks means that all rewrite rules specified with the **`rule`** expression are type-safe: applying a strategy where all rules are defined in this way is guaranteed to not suffer a reification error. This differs from some systems such as Stratego where type preservation is (optionally) checked at run time. This does mean that some applications of rewriting, such as translations from one terms of one language to terms of another, are less straightforward to implement with rewriting in Silver; doing so requires introducing additional productions to bridge between nonterminals of the different languages, that only exist during the rewriting process. However this is not a great loss since language translation and other such operations can typically be expressed more conveniently using higher-order attributes.

6.3. Congruence Traversal Strategies

In the lambda calculus example from Figure 23, the `evalInnermost` strategy on line 42 performs a full innermost rewrite over the entire tree. This is not a correct evaluation strategy since it also reduces inside abstraction bodies before they are applied to some argument. This may result in non-termination for otherwise-terminating expressions. More precise control over the traversal of the tree is needed than what is possible with generic traversal strategies like `all` and `one`. This is possible through the use of *congruence traversals*, also introduced by Stratego [24] and implemented in the Silver rewriting system.

Such strategies are built by the constructor **`traverse`** *`PROD_NAME(S1, S2, ...)`*, where *PROD_NAME* is the name of a production.[18] When applied to a term with a different production this strategy will always fail, otherwise it will apply each argument strategy to the corresponding child term, and if all succeed, construct a new term from the child results with the same production. For example, a lambda calculus term can be converted to weak head normal form with the following strategy, which does not normalize under abstractions:

```
1  global evalWHNF::Strategy =
2    try(traverse app(evalWHNF, evalWHNF) <+
3        traverse letT(id(), evalWHNF, evalWHNF)) <*
4    try((beta <+ letDist) <* evalWHNF);
```

[18] The initial **`traverse`** keyword is not present in Stratego's syntax; since strategies are constructed by ordinary expressions in Silver, this has been added to avoid an ambiguity with function/production application.

The congruence traversal strategy constructor is implemented by the `Strategy` production

```
traversal :: (Strategy ::= String [Strategy])
```

The implementation is somewhat similar to that of `all` in Figure 21, except that the name of the production and the particular strategy for each child is passed into the `AST` using inherited attributes.

6.4. Rewriting with Concrete Object-Language Syntax

Since the rewriting extension reuses Silver's `Pattern` and `Expr` nonterminals, object-language concrete syntax (as described in Section 5) can be used to specify rules without any additional implementation effort. An example of this can be found in Figure 27. Here term rewriting is used to normalize for-loops for the Halide extension to ableC that do not fit the exact form recognized by the pattern matching on line 11 of Figure 13. This normalization requires a sequence of strategies, which demonstrate a number of the previously-discussed features.

First, all loop expressions involving constants are simplified by the `simplifyLoopExpr` strategy (line 2 of Figure 27.) This uses a traversal strategy to apply the `simplifyExpr` strategy (lines 4—3) only within the loop expressions, and not within the loop bodies. The `preprocessLoop` strategy (lines 12—21) is used to fully expand operators such as `<=` such that the test and update expressions only involve `<`, `>`, `+`, `-` and `=`. Finally the loops are transformed by the `transLoop` strategy (lines 27—46) such that all loops range from 0 and increment by 1. This may involve introducing new local variables and renaming references to the loop variable accordingly. This is done using the `rename` strategy (line 25.)

6.5. Discussion

The original implementation of the ableC substitution library was defined using a `substituted` attribute and equations for every production in the ableC abstract syntax. These all follow the same pattern and were replaced by this term-rewriting-based implementation of substitution, replacing over 2,500 lines of boilerplate code [19] in the ableC specification (11.8% of the code base), in addition to numerous similar equations on ableC extension productions.

Unlike Stratego, our system does not support non-linear patterns in rewrite rules. This is primarily a limitation of ordinary pattern matching in Silver, the semantics of which are reused for specifying rules in the rewrite rule extension portion of the system; adding non-linear pattern support to the rewriting library would be straightforward. However this does not pose a particularly significant burden, as such cases can be dealt with using multiple variables and `when`-clauses, as seen for example on line 32 of Figure 23.

7. Reflection for Implementing Evaluators for Staged Languages

Here we provide a final example of the use of reflection in attribute grammars. We implement an evaluator for a simple staged programming language, a subset of MetaOCaml [25]. Staged programming is a programming paradigm in which fragments of code may be constructed at runtime, passed around as values, and eventually executed, all in a type-safe manner. Examples of such languages include MetaML [26] and MetaOCaml. This paradigm can provide performance improvements by generating a specialized piece of code that is used more than once. The canonical example in this area is a staged power function to efficiently compute an exponent x^n by generating code specialized for a given value of n; a MetaOCaml implementation of this is shown in Figure 28. Such languages use strong typing to allow well-defined semantics when running generated code [27]. Calcagno et al. [28] describes an approach to implementing staged languages by translation to a lower-level language containing constructs for reflection and term construction. One can also build a direct evaluator for a staged language, the approach we have chosen to demonstrate.

In addition to all the standard functional programming constructs, MetaOCaml has 3 new operators: quote (`.< >.`), escape (`.~`) and run (`.!`). Quote constructs a value of type `'a code`, where `'a` is the type of the quoted expression, corresponding to a fragment of code being constructed. For example, `.<1>.` on line 3 has the type `int code`. The escape operator can be written inside of a quoted expression, indicating that the escaped expression, when evaluated, will yield a piece of code that should be plugged into the result. The run operator executes

[19] See ableC/grammars/edu.umn.cs.melt.ableC/abstractsyntax/substitution in ableC 0.1.2

```
global simplifyLoopExpr:Strategy =
  traverse forDeclStmt(simplifyExpr, simplifyExpr, simplifyExpr, id());
global simplifyExpr::Strategy = innermost(simplifyExprStep);
global simplifyExprStep::Strategy =
  rule on Expr of
  -- Simplify expressions as much as possible
  | ableC_Expr { $Expr{intExpr(a)} + $Expr{intExpr(b)} } -> intExpr(a + b)
  ...
  | ableC_Expr { $Expr{intExpr(a)} / $Expr{intExpr(b)} }
    when b != 0 -> intExpr(a / b)
  end;
global preprocessLoop::Strategy =
  rule on Stmt of
  -- Normalize condition operators
  | ableC_Stmt {
      for ($Decl{init} $Name{i} <= $Expr{limit}; $Expr{iter}) $Stmt{b}
    } -> ableC_Stmt {
      for ($Decl{init} $Name{i} < $Expr{limit} + 1; $Expr{iter}) $Stmt{b}
    }
  ...
  end;

function rename
Strategy ::= n1::String n2::String
{ return topDown(try(rule on Name of name(n) when n == n1 -> name(n2) end)); }

global transLoop::Strategy =
  rule on Stmt of
  -- Normalize loops with nonstandard initial or step values
  | ableC_Stmt {
      for ($TypeExpr{t} $Name{i1} = $Expr{initial};
           $Name{i2} < $Expr{limit}; $Name{i3} += $Expr{step}) $Stmt{b}
    } when i1.name == i2.name && i1.name == i3.name ->
      let newName::String = freshVarName()
      in ableC_Stmt {
        for ($TypeExpr{t} $Name{i1} = 0;
             $Name{i2} < ($Expr{limit} - $Expr{initial}) / $Expr{step};
             $Name{i3}++) {
          typeof($Name{i1}) $name{newName} =
            $Expr{initial} + $Name{i1} * $Expr{step};
          $Stmt{rewriteWith(rename(i1.name, newName), b).fromJust}
        }
      }
      end
   ...
  end;
global normalizeLoop:Strategy =
  bottomUp(try(simplifyLoopExprs <* repeat(preprocessLoop))) <*
  topDown(try(transLoop <* simplifyLoopExprs));
```

Figure 27: Some of the rewrite rules used to normalize loops in the Halide extension. See ableC-halide/abstractsyntax/IterStmt.sv

```
1  nonterminal Expr with env, value;
2  synthesized attribute value::Value;
3
4  abstract production quoteExpr
5  top::Expr ::= e::Expr
6  { local a::AST = reflect(e);
7    a.env = top.env;
8    top.value =
9      codeValue(a.evalNestedEscapes);
10 }
11 abstract production escapeExpr
12 top::Expr ::= e::Expr
13 { top.value = error("undefined"); }
14 abstract production runExpr
15 top::Expr ::= e::Expr
16 { local trans::Expr =
17     case reify(case e.value of codeValue(a) -> a end) of
18     | right(e) -> e
19     | left(msg) -> error("Interpreter bug! " ++ msg)
20     end;
21   trans.env = top.env;
22   top.value = trans.value;
23 }
```

```
24 nonterminal Value;
25
26 abstract production intValue
27 top::Value ::= i::Integer
28
29 abstract production boolValue
30 top::Value ::= b::Boolean
31
32 abstract production codeValue
33 top::Value ::= a::AST
34
35 abstract production closureValue
36 top::Value ::=
37   id::String body::Expr
38   env::[Pair<String Value>]
```

Figure 29: A (simplified) portion of the implementation of a staged interpreter using reflection. Standard expression productions for operators, conditionals, lambda, etc. are not shown. Static type checking is also ignored, and `value` is actually monadic in the full implementation for run-time errors handling. See metaocaml/abstractsyntax/Value.sv and Expr.sv

a code value. Static typing ensures that no run-time type errors occur in evaluating quoted code, although run-time checking is required to ensure that some corner cases involving free variables are not evaluated. For example, `.<fun x -> .~(.! .<x>.)>.` is well-typed but in constructing this functional value the argument `x` to the function should not be evaluated, as would happen here.

The reflection system provides what is needed to elegantly implement our subset of MetaOCaml using attribute grammars[20]. A simplified portion of the implementation is shown in Figures 29 and 30. Phrases in the language are represented by the `Expr` nonterminal on line 1 in Figure 29. In addition to the usual productions for expressions (not shown), there are productions `quoteExpr`, `escapeExpr` and `runExpr`, each of which wraps a single expression. Evaluation is done by the synthesized `value` attribute of type `Value`, which is defined by the nonterminal defined on line 24. Values can integer values (`intValue`), closures (`closureValue`), or code values (`codeValue`) represented by an `AST` value. An inherited attribute `env` on `Expr` passes down the value environment mapping names to values in the expected way.

```
1 let square = fun x -> x * x
2 in let rec spower = fun n x ->
3   if n = 0 then .<1>.
4   else if n mod 2 = 0
5   then .<square .~(spower (n/2) x)>.
6   else .<.~x * .~(spower (n-1) x)>.
7 in let power7 =
8   .! .<fun x -> .~(spower 7 .<x>.)>.
9 in power7 2
```

Figure 28: An example use of staged programming to dynamically generate an efficient power function.

[20] Available at `http://melt.cs.umn.edu/` and `https://github.com/melt-umn/meta-ocaml-lite`, archived at `https://doi.org/10.13020/z10a-7g60`.

```
synthesized attribute evalNestedEscapes::Value occurs on AST;
attribute env occurs on AST;

aspect production nonterminalAST
top::AST ::= prodName::String children::ASTs
{
  local escape::Expr =
    case children of
    -- Only demanded when this is an escapeExpr
    | consAST(a, nilAST()) -> reify(a).fromRight
    end;
  escape.env = top.env;

  top.evalNestedEscapes =
    if prodName == "metaocaml:escapeExpr"
    then
      case escape.value of
      | codeValue(a) -> a
      | _ -> error("Escaped expression should evaluate to a code value")
      end
    else nonterminalAST(prodName, children.evalNestedEscapes);
}
```

Figure 30: The use of attributes on AST to evaluate contained escape productions. See metaocaml/abstractsyntax/Value.sv

To evaluate a quoted expression– the `e` child in `quoteExpr` on line 4 – the wrapped `Expr e` is first reflected. On the resulting `AST` we want to replace all `escapeExpr` constructs with the value of the code they contain. This done using a synthesized attribute `evalNestedEscapes` of type `Value`, declared on line 1 of Figure 30. If the `AST` is a reflection of an `escapeExpr` production (checked on line 15) then we get reified `Expr` that was wrapped up under `escapeExpr` (lines 7—11) and evaluate it using the environment (line 12). This will result in a `codeValue` (lines 17—18) value, which is stored in the attribute `evalNestedEscapes`. This process is somewhat similar to the previously described process of translating object-language literals or performing substitutions. Finally to evaluate a run expression, the operand is evaluated, the `AST` is extracted from the resulting `codeValue`, reified, and itself evaluated (lines 16—22 of Figure 29.) Note that this **`reify`** operation should always succeed as any ill-formed `code` values should have been caught by the MetaOCaml type checker.

This approach to implementing staged languages is primarily designed as a demonstration of the capabilities of the reflection system presented here, and is not intended to maximize performance as with more traditional compiled implementations [28]. We have not analyzed the difference in performance between these approaches.

Also of note is the recurring pattern of an escape production whose content is reified directly rather than being further analyzed. This is analogous to the antiquote productions encountered in Sections 5 and 6.2.

8. Reflection and Extensible Language Specifications

In Section 6, an earlier implementation of substitution using attributes was mentioned. This approach relied on what is called a *functor* transformation attribute named `substitutedResult` on all nonterminals. On each production is an equation for reconstructing the same tree using the value of `substitutedResult` on the children, except for those few productions for which the substitution is to be performed. Besides requiring a large amount of boilerplate specification, this approach is also flawed for a more subtle reason.

Consider the example in Figure 31, based on the template extension previously introduced in Figure 14. At the top of Figure 31 is shown a simplification of the `templateTypeExpr` production, which forwards to a reference

```
1  abstract production typedefTypeExpr
2  top::TypeExpr ::= n::String
3  { top.pp = n;
4    top.substitutedResult =
5      if n == top.substName then top.substTypeExpr else top;
6    ...
7  }
8  abstract production templateTypeExpr
9  top::TypeExpr ::= n::String args::TypeExprs
10 { top.pp = n ++ "<" ++ args.pp ++ ">";
11   top.substitutedResult = templateTypeExpr(n, args.substitutedResult);
12   forwards to typedefTypeExpr("_template_" ++ n ++ "_" ++ args.mangledName);
13 }
```

```
1 template<a> struct foo {            1 typedef struct {
2   a *ptr;                           2   int *ptr;
3 };                                  3 } _template_foo_int;
4 template<a> void f() {       ⟹     4 void _template_f_int() {
5   foo<a> x;                         5   _template_foo_int x; // foo<int> x;
6 }                                   6 }
7 ...                                 7 ...
8 f<int>();                           8 _template_f_int();
```

Figure 31: An example of the derived names problem occurring in the template extension. Top: simplified versions of the `typedefTypeExpr` and `templateTypeExpr` productions. Not shown is code to construct the instantiated implementation of the type and lift it to the global level. Bottom left: a fragment of code utilizing the template extension. Bottom right: the desired translation of this code fragment.

to a name, typedef'ed to the instantiated implementation of the type. For example, the code fragment in the lower left of the figure should forward to the fragment on the lower right. Note that the `templateTypeExpr` production forwards to a tree containing a type name *derived* from `args`. If a substitution is performed on `templateTypeExpr` by accessing `substitutedResult`, we wish to produce a `templateTypeExpr` parameterized by the substituted `args`. If instead we computed `substitutedResult` on the forwarded-to tree, we would incorrectly get a reference to `_template_foo_a`, an undefined type name. In the bottom of Figure 31 we see the correct result computed on the original forwarding tree which produces `_template_foo_int` on line 5. Thus we must provide the explicit `substitutedResult` equation defined on line 11 to get the desired behavior.

The modular well-definedness analysis, which checks that all attributes have defining equations, would not catch the need for this equation as an (incorrect) equation for this attribute is already provided by the forward. In fact the presence of this equation violates a non-interference notion of coherence [29], which states that attribute values defined by a production should be equivalent, in some semantic way, to the values obtained by forwarding. When all equations are coherent, composed extensions will not interfere with each other in unexpected ways (such as by causing errors in generated code); for this attribute we can make no such guarantees. We can, however, circumvent this problem by using reflection, as it operates on a syntactic level, before the introduction of semantic operations performed with attributes, as we have done above.

Another example of the trade-offs involved with syntactic vs. semantic operations may be seen with the ableC-Halide extension. Consider an independently-created **`forall`** extension that provides cleaner syntax for multiple nested loops (used on line 5 of Figure 32 along with the Halide `transform` extension). Let us assume that this **`forall`** construct forwards to the equivalent long-form for loops that are shown in the comment on line 4. Pattern matching in Silver takes forwarding into consideration; if a pattern in a `case` expression does not match the tree being inspected, then the pattern matching will be done on the tree that the inspected tree forwards to. Consequently loops specified with **`forall`** will be handled like any other for-loop by `transform` statements, since the pattern on lines 11—15 in Figure 13, will look through this new **`forall`** construct to see the host-language for-loops that it

```
1  void matmul(unsigned m, unsigned n, unsigned p,
2              float a[m][p], float b[p][n], float c[m][n]) {
3    transform {
4      // for (unsigned i = 0; i < m; i++) for (unsigned j = 0; j < n; j++) {
5      forall (unsigned i : m, unsigned j : n) {
6        c[i][j] = 0;
7        for (unsigned k = 0; k < p; k++) {
8          c[i][j] += a[i][k] * b[k][j];
9        }
10     }
11   } by { ... }
12 }
```

Figure 32: An example of the **forall** extension applied to the matrix multiplication example from Figure 12. A new statement (line 5) forwards to an equivalent set of nested loops (line 4.)

forwards to.

However, this nice behavior is rather brittle. Instead of translating loops as seen on line 4, **forall** could hypothetically forward to a slightly different tree, for example translating this as

```
for (unsigned i = 0; m > i; i += 1) for (unsigned j = 0; n > j; j += 1)
```

Such loops could ordinarily be handled by the rewriting described in Section 6.4. However in this scenario the ableC-Halide extension would not successfully rewrite and recognize the loops, as reflection (and thus term rewriting) operates syntactically on terms and thus does not respect forwarding.

These issues raise a more general question: if extensions can introduce new analyses (such as the above transformations) using reflection/rewriting instead of through attributes and forwarding, how can we ensure that these analyses will be well-defined [19] and non-interfering [29] when composed with other extensions? Well-definedness essentially boils down to ensuring that any new analysis always has some sensible default available, whether this is a forward equation in the case of attributes, or a general equation on `nonterminalAST` for any reflected production. Non-interference is concerned with the semantics of decorated trees, while reflection is typically applied to trees that are never directly decorated. However this isn't necessarily always the case; more research is needed to address this problem.

9. Implementing the Reflection System

This section provides a brief discussion of the implementation of the Silver reflection system. Silver is implemented by translation to Java. Basic types (such as strings and integers) all have concrete Java equivalents. For terms there is a Java abstract class `Node` that represents all nonterminals; every nonterminal type is represented by an abstract subclass of `Node`, and every production of a nonterminal is a concrete subclass of that nonterminal's class. In this way, the Java type system encodes the syntax of the object-language being defined in Silver. `Node` contains a number of abstract methods, such as one to get the Silver name of a production or iterate over its children. Since no extra type-level information is required, **reflect** is a Silver foreign function that calls a recursive Java function to walk a `Node`, calling the appropriate constructors corresponding to `AST` productions.

The implementation of **reify** is significantly more complex due to its run-time type-dependent nature; **reify** is thus a language construct in Silver (as opposed to being a foreign function as in the case of **reflect**). **reify** does run-time type checking in constructing a Silver tree to ensure that it is well-sorted. To do this, **reify** requires the run-time representation of the type of the tree it is to produce; this type is provided by Silver's type inference system. For `AST` trees representing integers or strings, the reification into a Silver-value of type `Integer` or `String` is straightforward. For a `nonterminalAST`-constructed tree, **reify** gets the production name (the first child of production `nonterminalAST`) and uses Java reflection to determine if that production exists. If so, gets the Java

Class object implementing that Silver production. On this object it then calls a static **reify** method (generated as a part of the class), parameterized by the expected return type it is to construct and the array of child AST trees yet to be reified. It checks that the appropriate number of children were provided, unifies the given expected type with the actual production left-hand-side nonterminal type, then reifies the children using the corresponding right-hand-side types. Finally, it constructs and returns a new production object with the results.

To support this, runtime type information, as an object of a new Java TypeRep class, is stored on each tree node. Because Silver supports parametric polymorphism, a runtime type unification process was also added, to mirror the compile-time type unification process. This is used when checking that types are compatible when constructing new trees. If not, **reify** returns an error. All Silver value classes implement a Typeable interface with a getType function returning a TypeRep value. Any type, *e.g.* Silver functions, or other new foreign type to be reflected into an AST tree, using the type-parametric anyAST production must implement this interface.

10. Related Work

Below we discuss related work on reflection and also work related to the examples uses of reflection described above: object-language concrete syntax, term-rewriting, and staged programming languages.

10.1. Reflection

Smith [30] introduced the notion of reflection and it has been widely used and discussed in many contexts in computer science. Demers and Malenfant [5] provide a more precise definition of reflection and also distinguish between *structural reflection* in which the language provides introspection of the program being executed and its abstract data types, and *behavioral reflection* in which the language can affect its own semantics. Our work is more closely related to structural reflection, as we are primarily focused on exposing data in Silver for generic introspection. A more formal view of reflection is given by Clavel and Meseguer [31] in which they define reflective languages as ones in which there is a mapping between certain data types in the language and a portion of the language's semantics.

Reflection is common in object-oriented languages such as Java [32]. In these contexts reflective operations are often tightly coupled to the objects under consideration. For example, it is possible to use reflection to get the value of a field based on a string representation of its name; this results in another object that can be cast to the expected type or to be reflected again. This differs from the notion of reflection used in this paper. Instead, a term composed of many nodes of various nonterminal types is reflected in its entirety into a distinct AST representation upon which further operations may be performed. In our experience this approach is a better fit for language meta-programming, as any operation on terms is primarily either *structural*, like serialization (and thus best implemented by attributes on AST), or *semantic*, like type checking (and thus best implemented by attributes on the nonterminals in question.)

10.2. Generic Programming using Type Classes and Object Algebras

A family of approaches to operating generically on well-sorted data are based on type classes. Numerous problems related to performing generic operations on algebraic/inductive datatypes, including substitution and serialization, are described in the scrap-your-boilerplate papers [4, 33]. This is mostly achieved through the use of a more sophisticated type system (*e.g.* type classes), but an approach based on reflection to a generic DataType representation is also described. This is similar to our AST, (representing inductive data type values, lists, constants etc.) however operations on DataType are expressed using recursive functions and type classes rather than attributes.

GHC.Generics [34] is another library for generic programming in Haskell, based around an automatically-derivable type class Generic that defines an alternative generic representation for algebraic data types, with constructors for sums, products, and type-level metadata. The Generic type class includes functions from and to for transforming between arbitrary values and this generic representation, similar to our **reify** and **reflect**. New generic operations (such as serialization) are defined as type classes with default method implementations that work on this generic representation. This representation differs from our approach as it only goes one level deep, consisting of generic sum and product constructors that ultimately wrap ordinary values corresponding to the children of a data constructor, while our approach deeply transforms a term to and from the AST generic representation all at once.

A powerful approach to representing data such as abstract syntax trees in object oriented languages is object algebras [35]. This is done by defining interfaces with methods corresponding to various data constructors, and classes

that implement these interfaces corresponding to various operations (such as evaluation or pretty-printing.) Like attribute grammars, this pattern allows for extensibility with both new data variants and new analyses by extending interfaces with additional constructor methods and defining additional implementation classes. Using this approach it is possible to express generic, scrap-your-boilerplate-style traversals and operations on abstract syntax trees. This is done in the Shy framework [36] by using Java 8 `default` methods in interfaces to provide default implementations of traversal operations for otherwise-untouched data variants.

10.3. Object-Language Concrete Syntax

As we have seen above, writing object-language concrete syntax in a meta-language with quote and antiquote operators dramatically simplifies language specifications by avoiding a tremendous amount of boilerplate code. This, of course, has been noted previously and seen in tools such as ASF+SDF [37] and Stratego [24, 7]. The approach used in Stratego is particularly relevant here and how it it differs from our approach. In Stratego the transformation from embedded object-language abstract syntax to meta-language abstract syntax is done incrementally through term rewriting, during which ill-sorted intermediate trees composed of both meta- and object-language abstract syntax exist. Thus, ill-sorted trees must be represented. The TypeSmart feature in Stratego [38] can (optionally) be used to dynamically disallow ill-sorted trees; when a constructor is applied TypeSmart checks that the arguments are of the required sort. However TypeSmart is not compatible with Stratego's object-language to meta-language translation strategies [39].

In our reflection-based approach, dynamic type checking is only needed in the limited use of **`reify`** for handling antiquote productions, since the rest of the translation process is checked statically. This comes at the cost of the added complexity of a separate `AST` representation, not required in a rewriting approach. Instead of having the **`reify`** operation dynamically type-check the `AST`, we could have the `AST` constructors check the types of their represented argument values, as done with the TypeSmart feature of Stratego. However this would require a mechanism to access (in Silver specifications, not just in its runtime as **`reify`** does) the types of productions by name.

Squid quasiquotes [40, 41] provide concrete object-language syntax embedding for syntax tree construction in Scala. These bear some resemblance to our `AST`-based approach, most notably by allowing introspection via pattern matching of quoted `Code` values, but Squid provides stronger type safety guarantees about quoted fragments by tracking their types as in staged languages. Conversely, the variety of type systems used by object languages implemented in Silver effectively preclude a similar approach to static type safety in concrete syntax patterns and literals.

10.4. Term Rewriting

The term rewriting system presented is very similar to Stratego, providing many of the same primitive and library strategy constructors. It is in many ways an implementation of Stratego features in an attribute grammar setting. The primary differences arise because Stratego is a standalone language, while our reflection-based approach is embedded within Silver. This allows for the use of Silver features in the definitions of rules and strategies. For example, strategies can be created as local variables within functions or productions, and expressions on the right sides of rules can use parameters of the enclosing function or access attributes. These features arguably provide a cleaner alternative to some of Stratego's, such as dynamic rule creation.

Another significant difference from Stratego is that (as described at the end of Section 6.2) in our system all rules defined using the rewriting extension are statically type-checked, instead of optionally at run-time with TypeSmart. Static type checking is generally less error-prone but at the cost of flexibility; some rewriting idioms, such as translations from one sort of tree to another, would require additional boilerplate productions bridging between different sorts of trees. However such translation rewrites can arguably be done in a cleaner and safer way using attributes.

TOM [42] is an extension to Java for algebraic data types and term rewriting that bears some similarities to our system. In TOM strategies are built from rules and applied to terms using features of the Java host language; the right side of a rule is ordinary Java code in much the same way as the right side of a rule in Silver is an ordinary Silver expression. Additionally all rules are statically checked and guaranteed to be type-preserving. However the implementation differs as instead of operating on a separate, generic term representation, rules work directly on the well-sorted representation. The generic view needed for traversal combinators such as `All` is provided by the object-oriented nature of the language, with algebraic data types being compiled into classes sharing a common superclass.

JastAdd [2] is another attribute grammar system that integrates features for rewriting [43]. However these features differ significantly from rewriting in Silver and other strategic rewriting systems. In JastAdd rewriting is done on

trees that have already been decorated with attributes (as opposed to terms in our approach), and all rewrite rules are applied in a global, innermost order with no notion of strategies to control the order of rewriting. The use of rewriting in JastAdd also differs significantly from rewriting in Silver in that it is more closely related to *forwarding* [8] in Silver. The typical use-case for both is to translate language constructs down to some canonical or core form in the language. In Silver this is used to translate language extension constructs into the representation in a host language. In the JastAdd extensible Java compiler [44], the rewriting is used to specialize different uses of the generic dot-notation (*e.g.* `A.b`) into forms specific to the cases when item being accessed (`A`) is a package, class, or object. This depends on a typing context and thus attributes must be present to guide the rewriting. Both of these use cases can be carried out in either tool since both use attribute values in the transformation and the transformation is carried out automatically; there is no specific operation to forward or rewrite that can be invoked. In JastAdd this rewriting is done when an attribute is first accessed on a tree, causing the transformation to be made. The rewritten tree then replaces the original and attribute values are retrieved from it. With forwarding, the original tree is *not replaced* by the transformed (forwarded-to) tree; instead the original forwarding tree has a link to the new tree.

Kiama [1] is a language processing system written as an embedded DSL in Scala that incorporates both attribute grammars and term rewriting. Of special interest here is that Kiama also takes inspiration from Stratego and provides strategy combinators like those discussed above. Since Scala is also strongly and statically typed, Kiama also needs some way to manipulate well-sorted syntax trees in a generic manner. This is primarily achieved by the sub-classing afforded by the object-oriented nature of Scala. In Kiama productions are represented using *case-classes*, a form of sub-classing that exposes the structure of a tree so that pattern matching can be performed. A nonterminal type is thus a sub-class of the *product* type in Scala and this type provides generic access to the name of the tree constructor and the list of its children. This provides an interface not unlike that of the `nonterminalAST` production in our `AST` representation as seen on line 4 in Figure 5. Thus for most activities, there is no need for reflection. It is used in one way however. The Scala copy-constructor, which would be used to reconstruct a tree with its rewritten children, cannot be used in a generic manner and thus a bit of Java reflection is used in this instance to get access to the constructor class and to rebuild the tree with a generic constructor in the reflection library.

We have recently developed an alternative notion of rewriting in Silver known as *strategy attributes* [45] that operates on trees that have been decorated with attributes rather than on undecorated terms. This approach works by translating strategy expressions into higher-order attributes and equations instead of using reflection. Doing so allows contextual information in the form of inherited attributes to be utilized in rewrite rules, and forwarding is respected, addressing issues with the use of reflection-based rewriting as outlined with the ableC-Halide extension in Section 8. Additionally this approach of statically compiling strategies seems to have better performance than the reflection-based implementation presented here, however the compiled approach somewhat limits flexibility in comparison to reflection, as strategies can no longer be constructed dynamically.

10.5. Staged Programming Languages

A definition of staged programming and the MetaML language is given by Sheard [26]. Calcagno et al. [28] introduces the MetaOCaml language and describes an approach to implementing staged languages by translation to a target language containing constructs for reflection and abstract syntax tree manipulation. This differs from our design, in which a meta-language supplying such constructs is used to build a direct interpreter for evaluating a staged language.

Our approach is primarily motivated as a demonstration of the capabilities of reflection, and is likely not appropriate for applications where performance is a primary concern. Other implementations of staged languages, such as the compilation approach described by Calcagno et al. [28] are probably much more efficient, however we have not done any performance analysis.

11. Discussion and Conclusion

This paper integrates a form of reflection into attribute grammars with a **`reflect`** construct of turning well-sorted terms (syntax trees yet to decorated with attribute values) into a generic representation and an inverse **`reify`** operation. In meta-programming systems, such as Silver, in which the type system of the the specification language enforces the well-sortedness of terms, writing analyses and transformations of such trees can involve writing a significant amount

of boilerplate specifications. We have shown how reflection lets the language developer reflect a term into a generic form, process or analyze it in a convenient generic way, and then, in some applications, reify the tree back to the well-sorted form. In the example applications, many lines of boilerplate Silver specification were eliminated from existing applications or were avoided from the beginning. In our experience, even though well-sortedness is not guaranteed by the type system in the `AST` form we rarely found this to be problem; it is easily outweighed by the savings in lines of specifications written. However, we recognize that this evaluation may not be case for all users, especially those new to Silver and reflection.

There are many additional uses of reflection in attribute grammars beyond those discussed here. Examples include writing visitor-pattern-style traversals over trees, mechanisms for (runtime) type-safe casts, and implementation of generic map and reduce operations over arbitrary trees. Reflection opened up the possibility of building the term-rewriting extension to Silver, something that was not covered in the original conference version of this paper [9]. One area of future work includes improving the performance of the processing of Silver interface files by replacing the text-based serialization system with one that generates more compact binary representations.

Another potential future area of improvement involves the type-checking behavior of **`reify`**. It is not possible for a library function (such as for de-serialization) to construct a result with polymorphic type by using **`reify`**, as the expected type against which to check the `AST` depends on the context in which the function is called, and cannot be determined statically. Instead such operations must be implemented as language extensions to Silver, such that the call to **`reify`** in their translation is type checked and translated specifically for the desired type. This could be avoided by introducing a built-in type class to track the specific run-time types with which polymorphic functions are called.

The design and applications of reflection presented here, while presented in the context of attribute grammars, are not limited to such. Other functional languages (such as ML, that lacks type classes needed for "scrap-your-boilerplate"-style generic programming [4]) could benefit from a similar approach of transforming to and from a generic view of data. In such a scenario the applications presented here, such as serialization and de-serialization, could be translated rather directly with attributes on `AST` being replaced by recursive functions.

Acknowledgments

We thank Anthony Sloane for our discussions about Kiama and its approach to term-rewriting that helped us to better understand the approach taken there. We also thank the anonymous reviewers of this paper for many helpful comments and suggestions.

References

[1] A. M. Sloane, Lightweight language processing in Kiama, in: Proceedings of the 3rd summer school on Generative and Transformational Techniques in Software Engineering III (GTTSE '09), Vol. 6491 of Lecture Notes in Computer Science, Springer, 2011, pp. 408–425. doi:10.1007/978-3-642-18023-1_12.

[2] T. Ekman, G. Hedin, The JastAdd system - modular extensible compiler construction, Science of Computer Programming 69 (2007) 14–26. doi:10.1016/j.scico.2007.02.003.

[3] E. Van Wyk, D. Bodin, J. Gao, L. Krishnan, Silver: an extensible attribute grammar system, Science of Computer Programming 75 (1–2) (2010) 39–54. doi:10.1016/j.scico.2009.07.004.

[4] R. Lämmel, S. P. Jones, Scrap your boilerplate: A practical design pattern for generic programming, in: Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI), ACM, 2003, pp. 26–37. doi:10.1145/604174.604179.

[5] F.-N. Demers, J. Malenfant, Reflection in logic, functional and object-oriented programming: a short comparative study, in: Proceedings of the IJCAII'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI, 1995, pp. 29–38.

[6] T. Kaminski, L. Kramer, T. Carlson, E. Van Wyk, Reliable and automatic composition of language extensions to C: The ableC extensible language framework, Proceedings of the ACM on Programming Languages 1 (OOPSLA) (2017) 98:1–98:29. doi:10.1145/3138224.

[7] E. Visser, Meta-programming with concrete object syntax, in: Proceedings of the ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE), Vol. 2487 of Lecture Notes in Computer Science, Springer, 2002, pp. 299–315. doi:10.1007/3-540-45821-2_19.

[8] E. Van Wyk, O. de Moor, K. Backhouse, P. Kwiatkowski, Forwarding in attribute grammars for modular language design, in: Proceedings of the Conference on Compiler Construction (CC), Vol. 2304 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 128–142. doi:10.1007/3-540-45937-5_11.

[9] L. Kramer, T. Kaminski, E. Van Wyk, Reflection in attribute grammars, in: Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts & Experience (GPCE), ACM, 2019, pp. 48–60. doi:10.1145/3357765.3359517.

[10] E. Dolstra, E. Visser, Building interpreters with rewriting strategies, Electronic Notes in Theoretical Computer Science 65 (3) (2002) 57–76.

[11] T. Kaminski, L. Kramer, T. Carlson, E. Van Wyk, Reliable and automatic composition of language extensions to C — supplemental material, Tech. Rep. 17-009, University of Minnesota, Department of Computer Science and Engineering, available at https://www.cs.umn.edu/research/technical_reports/view/17-009 (2017).

[12] D. E. Knuth, Semantics of context-free languages, Mathematical Systems Theory 2 (2) (1968) 127–145, corrections in **5**(1971) pp. 95–96. doi:10.1007/BF01692511.

[13] E. Van Wyk, A. Schwerdfeger, Context-aware scanning for parsing extensible languages, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), ACM, New York, NY, USA, 2007, pp. 63–72. doi:10.1145/1289971.1289983.

[14] H. H. Vogt, S. D. Swierstra, M. F. Kuiper, Higher order attribute grammars, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, 1989, pp. 131–145. doi:10.1145/73141.74830.

[15] G. Hedin, Reference attribute grammars, Informatica 24 (3) (2000) 301–317.

[16] J. T. Boyland, Remote attribute grammars, Journal of the ACM 52 (4) (2005) 627–687. doi:10.1145/1082036.1082042.

[17] U. Kastens, Ordered attributed grammars, Acta Informatica 13 (1980) 229–256. doi:10.1007/BF00288644.

[18] M. Jourdan, An optimal-time recursive evaluator for attribute grammars, in: Proceedings of 6h International Symposium on Programming, Vol. 167 of Lecture Notes in Computer Science, Springer, 1984, pp. 167–178. doi:10.1007/3-540-12925-1_37.

[19] T. Kaminski, E. Van Wyk, Modular well-definedness analysis for attribute grammars, in: Proceedings of the 5th International Conference on Software Language Engineering (SLE), Vol. 7745 of Lecture Notes in Computer Science, Springer, 2012, pp. 352–371. doi:10.1007/978-3-642-36089-3_20.

[20] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, S. Amarasinghe, Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, New York, NY, USA, 2013, pp. 519–530. doi:10.1145/2491956.2462176.

[21] A. Schwerdfeger, E. Van Wyk, Verifiable composition of deterministic grammars, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, New York, NY, USA, 2009, pp. 199–210. doi:10.1145/1542476.1542499.

[22] T. Kaminski, E. Van Wyk, A modular specification of Oberon0 using the silver attribute grammar system, Science of Computer Programming 114 (2015) 33–44. doi:10.1016/j.scico.2015.10.009.

[23] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, Cambridge, U.K., 1998. doi:10.1017/CBO9781139172752.

[24] E. Visser, Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5, in: A. Middeldorp (Ed.), Rewriting Techniques and Applications (RTA'01), Vol. 2051 of Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 357–361. doi:10.1007/3-540-45127-7_27.

[25] O. Kiselyov, Reconciling abstraction with high performance: A MetaOCaml approach, Foundations and Trends in Programming Languages 5 (1) (2018) 1–101. doi:10.1561/2500000038.

[26] T. Sheard, Using MetaML: A staged programming language, in: International School on Advanced Functional Programming (AFP), Vol. 1608 of Lecture Notes in Computer Science, Springer, 1998, pp. 207–239. doi:10.1007/10704973_5.

[27] A. Filinski, A semantic account of type-directed partial evaluation, in: International Conference on Principles and Practice of Declarative Programming (PPDP), Vol. 1702 of Lecture Notes in Computer Science, Springer, 1999, pp. 378–395. doi:10.1007/10704567_2.

[28] C. Calcagno, W. Taha, L. Huang, X. Leroy, Implementing multi-stage languages using ASTs, Gensym, and reflection, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), Vol. 2830 of Lecture Notes in Computer Science, Springer, 2003, pp. 57–76. doi:10.1007/978-3-540-39815-8_4.

[29] T. Kaminski, E. Van Wyk, Ensuring non-interference of composable language extensions, in: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE), ACM, 2017, pp. 163–174. doi:10.1145/3136014.3136023.

[30] B. C. Smith, Procedural reflection in programming languages, Ph.D. thesis, Massachusetts Institute of Technology (1982).

[31] M. Clavel, J. Meseguer, Axiomatizing reflective logics and languages, in: Proceedings of Reflection, 1996, pp. 263–288.

[32] G. Kirby, R. Morrison, D. Stemple, Linguistic reflection in Java, Software: Practice and Experience 28 (10) (1998) 1045–1077. doi:10.1002/(SICI)1097-024X(199808)28:10<1045::AID-SPE191>3.0.CO;2-F.

[33] R. Lämmel, S. P. Jones, Scrap more boilerplate: Reflection, zips, and generalised casts, in: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP), ACM, 2004, pp. 244–255. doi:10.1145/1016850.1016883.

[34] J. P. Magalhães, A. Dijkstra, J. Jeuring, A. Löh, A generic deriving mechanism for Haskell, SIGPLAN Not. 45 (11) (2010) 3748. doi:10.1145/2088456.1863529.

[35] B. C. d. S. Oliveira, W. R. Cook, Extensibility for the masses, in: J. Noble (Ed.), Proceedings of the European Confernece on Object Oriented Programming (ECOOP), Vol. 7313 of Lecture Notes in Computer Science, Springer, 2012, pp. 2–27. doi:10.1007/978-3-642-31057-7_2.

[36] H. Zhang, Z. Chu, B. C. d. S. Oliveira, T. v. d. Storm, Scrap your boilerplate with object algebras, in: Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM, New York, NY, USA, 2015, p. 127146. doi:10.1145/2814270.2814279.

[37] M. G. J. v. d. Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, J. Visser, The ASF+SDF meta-environment: A component-based language development environment, in: Proceedings of the Conference on Compiler Construction (CC), Vol. 2027 of Lecture Notes in Computer Science, Springer, 2001, pp. 365–370. doi:10.1016/S1571-0661(04)80917-4.

[38] S. Erdweg, V. Vergu, M. Mezini, E. Visser, Modular specification and dynamic enforcement of syntactic language constraints when generating code, in: Proceedings of the 13th International Conference on Modularity, ACM, 2014, pp. 241–252. doi:10.1145/2577080.2577089.

[39] E. Visser, Personal communication. (2019).

[40] L. Parreaux, A. Shaikhha, C. E. Koch, Squid: Type-safe, hygienic, and reusable quasiquotes, in: Proceedings of the ACM SIGPLAN

International Symposium on Scala, ACM, New York, NY, USA, 2017, pp. 56–66. doi:10.1145/3136000.3136005.

[41] L. Parreaux, A. Voizard, A. Shaikhha, C. E. Koch, Unifying analytic and statically-typed quasiquotes, Proceedings of the ACM on Programming Languages 2 (POPL) (2017) 13:1–13:33. doi:10.1145/3158101.

[42] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, A. Reilles, Tom: Piggybacking rewriting on Java, in: F. Baader (Ed.), Term Rewriting and Applications, Vol. 4533 of Lecture Notes in Computer Science, Springer, 2007, pp. 36–47. doi:10.1007/978-3-540-73449-9_5.

[43] T. Ekman, G. Hedin, Rewritable reference attributed grammars., in: Proceedings of the 18th European Confernece on Object Oriented Programming (ECOOP), Vol. 3086 of Lecture Notes in Computer Science, Springer, 2004, pp. 144–169. doi:10.1007/978-3-540-24851-4_7.

[44] T. Ekman, G. Hedin, The JastAdd extensible Java compiler, in: Proceedings of the 22nd annual ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications (OOPSLA), ACM, 2007, pp. 1–18. doi:10.1145/1297027.1297029.

[45] L. Kramer, E. Van Wyk, Strategic tree rewriting in attribute grammars, in: Proceedings of the International Conference on Software Language Engineering (SLE), 2020, to appear. doi:10.1145/3426425.3426943.