Silver: an Extensible Attribute Grammar System

Eric Van Wyk, Derek Bodin, Jimin Gao, Lijesh Krishnan

Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN, USA

Abstract

Attribute grammar specification languages, like many domain-specific languages, offer significant advantages to their users, such as high-level declarative constructs and domain-specific analyses. Despite these advantages, attribute grammars are often not adopted to the degree that their proponents envision. One practical obstacle to their adoption is a perceived lack of both domain-specific and general purpose language features needed to address the many different aspects of a problem. Here we describe Silver, an extensible attribute grammar specification system, and show how it can be extended with general purpose features such as pattern matching and domain-specific features such as collection attributes and constructs for supporting data-flow analysis of imperative programs. The result is an attribute grammar specification language with a rich set of language features. Silver is implemented in itself by a Silver attribute grammar and utilizes forwarding to implement the extensions in a cost-effective manner.

Key words: extensible languages, extensible compilers, attribute grammars, forwarding, Silver attribute grammar system

1 Introduction

Domain-specific languages offer several significant advantages to their users over general purpose programming languages [11]. They allow problem solutions to be expressed using the notational constructs of the problem domain. These languages are often declarative in nature, resulting in concise programs. Also, important optimizations and analysis are often only feasible when the domain-specific information is directly represented in the language constructs of the DSL as opposed to encoding them in lower-level constructs in a general purpose language.

But, domain-specific languages have some disadvantages as well. Van Deursen et al. [11, page 27] describe several and we quote three that pose particular challenges to DSL implementers here:

- "The costs of designing, implementing and maintaining a DSL."
- "The difficulty in finding the proper scope for a DSL."
- "The difficulty of balancing between domain-specificity and general purpose programming language constructs."

Although many DSLs are widely used, these disadvantages (and others) sometimes prohibit the level of adoption envisioned by the DSL implementers.

In the domain of language analysis and translation, attribute grammar specification languages offer many advantages but are also not as widely used as they might be. Attribute grammars (AG) were developed almost 40 years ago by Knuth [28] and there has been a steady stream of research in such systems since then, see [50,14,5] to cite just a very few. The continued interest is due to the fact that they provide a high-level, declarative means for solving a wide variety of language analysis and translation problems. Evidence of this can be seen in their use in implementing language processing tools for full-fledged popular languages such as Java 1.5 [15,13] and Icon [22].

Our experience using attribute grammars is primarily with our own system, Silver. We have developed an attribute grammar specification language called Silver to incorporate an extension to AGs called forwarding [45] that has proven useful in the specification of extensible programming and modeling/specification languages. We have used Silver to specify an extensible implementation of Java 1.4 [47] and several modular language extensions. One embeds SQL into Java and performs static type checking of the embedded SQL queries [44]. We have also built an extensible version of (a substantial subset of) the synchronous language Lustre (used in embedded safety-critical systems) and various language extensions [21] for it.

In the early stages of this work, using a prototype implementation of Silver we found the challenges described by van Deursen et al. [11] and listed above to ring especially true. For example, we found situations where we wanted some of the general purpose features we enjoy in modern functional languages such as parametric polymorphism and pattern matching. We wanted features sometimes found in other AG systems like collections [5] or autocopy rules for inherited attributes to reduce boilerplate AG specifications. We also wanted additional features for specific problem domains addressed by AGs: performing data-flow analysis on imperative programs, for example. In our prototype attribute grammar implementations of Silver we found that we had created languages that were quite useful for problems that fit completely in the language's application domain but that felt brittle and overly constraining for

aspects of the applications that did not fit squarely in the traditional domain of attribute grammars. This view is not that uncommon and others [10, page 185] have noted that AGs can sometimes feel cumbersome and restrictive when compared to modern languages. Thus the primary challenges in implementing Silver were in determining what domain-specific and general purpose features should be included, and then implementing them in a cost-effective manner.

These are similar to the challenges that extensible languages are designed to address: lack of features, ease of implementation, modularity so that sets of features can be easily composed to create new languages (and their processing tools) that have the features needed to better address the problems faced by users of such languages. We thus decided to implement Silver as an extensible language in order to mitigate some these challenges. Through a series of bootstrapping steps we were able to implement Silver as an AG specification written in Silver.

The remainder of this section describes extensible languages as they are processed by Silver and outlines its development. Section 2 describes the features in the "core" Silver language and provides an example specification of a small imperative language to motivate the extensions to Silver that are described in Section 3. Section 4 discusses related work and Section 5 concludes.

1.1 Extensible Languages

In the extensible languages paradigm, languages are not treated as monolithic entities and implemented as such. Instead, new language features are implemented and deployed as modular *language extensions* that are added later, perhaps by the language user, to a *host language*. In the case of DSLs, the host language defines the core, fundamental features of the DSL. The language extensions define the desired language features that are not implemented in the host language. In our approach, the host language is implemented as an AG specification and language extensions are implemented as AG fragments. Language extensions may introduce new language constructs (notations), new semantic analyses that, for example, perform some error checking, or new translations to different target languages. A key characteristic of the language extensions that are supported is that new language constructs need to be translated to semantically equivalent constructs in the host language. Thus the host language must satisfy some notion of completeness.

Many extension constructs are implemented as *local transformations* that translate the extension construct to semantically equivalent constructs in the host language. This provides an implicit specification of the semantics (that is attributes) of the extension construct. This is done via *forwarding* [45] which

also allows *explicit* specification of semantics (attributes) at the extension language level.

Some language features cannot be implemented by purely local transformations but instead require non-local transformations. We are especially interested in *composable* language extensions; these are extensions that can be used in conjunction with other language extensions typically designed without knowledge of one another. One characteristic of composable extensions is that the order in which different constructs defined in different language extensions are translated down to host language constructs should not matter. Silver does not provide linguistic support for the sort of global transformations that cause radical rewrites of the original syntax tree. [1] Constructs that employ a certain type of global transformations for translation to the host language can be easily composed, however, if they satisfy two requirements. First, the global transformation for construct c in a program p to program p_H in the host language must be strictly *additive*; that is, new constructs may be added on a global scale in creating p_H but these do not involve a radical reorganization of p's global structure. Second, the constructs added to p_H cannot conflict with global additions made by other features. Two transformations that add new declarations to the beginning of a program to support the local transformations satisfy these requirements. Our attribute grammar-based methodology uses (higher-order) collection attributes and forwarding on key productions in the core language specification to enable the addition of new constructs on a global scale.

1.2 Development of Silver as an Extensible Language

A core attribute grammar language serves as the host language for the full-featured version of Silver that is used in specifying extensible programming and modeling languages and their extensions [47,21]. In addition to the traditional constructs introduced by Knuth [28] the core Silver language includes *higher-order* attributes [50] that allow attributes to store (undecorated) syntax trees. This is useful for creating new trees in building, for example, optimized versions of a program or for constructing data structures such as symbol tables or representations of types used for type checking. To support interesting language extensions, the core host Silver language must be Turing complete and thus higher-order attributes are essential. Attributes containing decorated trees are also allowed. These are essentially the same as Hedin's reference attributes [14] which allows attributes to store references (pointers) to nodes

[1] However, if one is willing to specify an order in which the global transformations of different extensions are to be made, then one can use higher-order attributes in Silver to implement the global restructuring to construct a new transformed tree.

in the tree. These are useful for linking variable uses to their declarations in various languages. The core language also includes forwarding [45], a feature that allows productions to implicitly define the value of attributes by translation. *Aspect productions* allow new attributes to be defined for an existing production typically defined in a different grammar module or file. Core Silver also has a module system used in composing host language and extension specifications. Section 2.1 discusses core Silver.

Several general purpose and domain-specific language extensions have been made to core Silver to create the full-featured version. These include pattern matching on trees (by production), type-safe polymorphic lists, and convenience constructs such as autocopy inherited attributes. We also implement collection attributes [5] as a language extension. These can be used in the AG specification of the host language to enable certain useful, but limited, global transformations that do not interfere with similar global transformations specified by other language extensions. Additional extensions provide constructs for building control flow graphs for imperative programs and performing data-flow analysis via model checking [46]. These extensions are discussed in Section 2.2 and 3. We will not provide formal definitions of attribute grammars [28], higher-order attributes [50], forwarding [45], or collection attributes [5] but will instead describe their functionality through examples. Formal descriptions can be found in the cited papers.

The end result is that Silver is an extensible full-featured attribute grammar specification language with many domain-specific and general purpose language features; it is constructed from a simple core AG language and composable, modular language extensions.

2 Silver attribute grammar specification language

In this section we describe the language features in core Silver and the general purpose and domain-specific features added as language extensions. We describe and motivate several of these features by providing a partial specification of a small C-like imperative language named SimpleC written in the full, extended Silver language.

2.1 Core Silver

To support the modular development of language specifications, attribute grammar specifications written in Silver can be distributed across different grammar modules. A Silver grammar module contains AG declarations and

definitions for nonterminals, terminals, productions, and attributes. Silver module names, like Java packages, are based on Internet domain names in order to avoid name clashes. Module names indicate directories, not files, and the implementation of a Silver module may be spread across several files in the specified directory. The scope of a construct defined in a particular file includes all of that file and all other files in the same module. Silver specifications can also define concrete syntax for a language that is used to generate a parser and scanner for the language. Fig. 1 shows part of the specification of the abstract syntax and its attributes for SimpleC. It is written primarily using constructs from core Silver. Fig. 2 shows a specification of its concrete syntax and also uses constructs from core Silver. Fig. 3 has additional abstract syntax specifications for type checking and utilizes several Silver features added as language extensions.

These figures correspond to different files in the `edu:umn:cs:melt:simplec` grammar. Additional files complete the definition of SimpleC but are not shown as the key characteristics of Silver can be seen in the figures presented.

Each Silver file begins with a *declaration* of the grammar name, as can be seen in the figures in the `grammar` declaration. After the grammar declaration (and any `import` statements that include AG declarations from other grammar modules) a Silver file consists of a series of AG declarations. Line comments begin with "`--`".

2.1.1 Silver specifications for abstract syntax

The essence of attribute grammars consists of specifications of nonterminals and productions that define a context-free grammar, and the declaration of attributes that decorate specified nonterminals, and the definitions for computing their value on different productions. To understand the language features in core Silver consider the specifications in Fig. 1. Reading from the beginning of that figure we see the declaration of nonterminal symbols `Prog` (program), `Dcl` (declaration), `Dcls`, `Stmt` (statement), `Expr` (expression), and `Type` (type expressions).

Next a *synthesized* attribute `c` of type `String` is declared. Synthesized attributes are used to propagate semantic information up the syntax tree. The attribute defines the translation of SimpleC to C and decorates the nonterminals specified in the `occurs on` specification. Other attributes and nonterminals referenced in this figure are defined in different files in the directory of files that define this grammar. Figs. 1-3 each show a portion of a single file in this directory.

Following are a few sample *production* declarations. Productions with the `abstract` modifier are not used to generate the input specification to a parser

```
grammar edu:umn:cs:melt:simplec;
nonterminal Prog, Dcls, Dcl, Stmt, Expr, Type ;
synthesized attribute c :: String ;
attribute c occurs on Prog, Dcls, Dcl, Stmt, Expr, Type ;

abstract production program   p::Prog ::= f::Dcls
{ p.c = "#include <stdio.h> \n" ++ f.c ;
  p.errors := f.errors ;
  f.env = [ :: Binding ] ;  }

abstract production while w::Stmt ::= cond::Expr body::Stmt
{ w.c = "while ( " ++ cond.c ++ ") \n" ++ body.c ;
  w.errors := cond.errors ++ body.errors ;     }

abstract production for
f::Stmt ::= init::Stmt cond::Expr inc::Stmt body::Stmt
{ forwards to stmt_seq (init, while(cond, stmt_seq(body,inc))) ; }

abstract production stmt_seq   s::Stmt ::= s1::Stmt s2::Stmt
{ s.c = s1.c ++ s2.c ; }

abstract production logical_and  e::Expr ::= l::Expr r::Expr
{ e.c = "(" ++ l.c ++ " && " ++ r.c ++ ")";
  e.errors := ... ;
  e.typerep = booleanType(); }

abstract production logical_not  e::Expr ::= ce::Expr
{ e.c = "( ! " ++ ce.c ")";
  e.errors := ... ;
  e.typerep = booleanType(); }

abstract production logical_or   e::Expr ::= l::Expr r::Expr
{ e.typerep = booleanType();
  e.errors := ... ;
  forwards to logical_not ( logical_and (logical_not(l),
                                         logical_not(r)));
  --  l || r becomes  ! (! l && ! r)
}

abstract production func_call  e::Expr ::= f::Id_t arg::Expr
{ e.c = f.lexeme ++ "(" ++ arg.c ++ ")" ;
  e.errors := arg.errors; }
```

Fig. 1. Abstract syntax specification for SimpleC.

generator. We will later see productions with the `concrete` modifier instead; these are used in constructing the parser. The first production is named `program`, its left-hand side nonterminal is `Prog` and is named `p`. In Silver nonterminals and terminals are named so that they can be referenced in the attribute definitions. The *has-type* construct `::` specifies the name of a nonterminal or terminal on its left and the type of the nonterminal or terminal on its right. The production's right-hand side contains the `Dcls` nonterminal named `f`. Programs in SimpleC are sequences of (function) declarations (`Dcls`). Productions that define this sequence and function declarations are not shown for space reasons. Attribute definitions are given between the curly braces ("{" and "}"). Here, the attribute `c` on `p` is defined to be a C include specification followed by the translation to C of the sequence of declarations (`f.c`). String concatenation is specified by the operator `++`. Definitions of other attributes that use features added as language extensions such as lists (`[...]`) and collections (`:=`) are also shown but described below in Section 2.2. For example, the inherited attribute `env` is defined in the `program` production for `f` to be the empty list.

Productions for loops, statement sequences and a few sample expression productions such as conjunction and negation follow. These define the attribute `c` in the expected ways. For example, in the *while* loop production named `while` the C translation attribute is constructed from the translations of the condition (`cond.c`) and the body (`body.c`). Its `errors` attribute is defined by concatenating the lists of errors from these components using the overloaded operator `++` that is also used for list concatenation.

The `for` and `logical_or` productions use *forwarding* [45] to implement local transformations that map these constructs to other semantically equivalent constructs in SimpleC. In the case of the for-loop, the `forwards to` clause specifies that *for* (*init; cond; inc*) *body* is equivalent to *init ; while* (*cond*) { *body; inc;* } . For logical-or it specifies that $l \vee r$ is equivalent to $\neg(\neg\, l \wedge \neg\, r)$. Forwarding allows a production to define a distinguished syntax tree that provides default values for synthesized attributes that it does not explicitly define with an attribute definition. When a tree node is queried for an attribute that is not explicitly defined, it "forwards" that query to this tree which will return its value. In `logical_or` this tree is the semantically equivalent expression constructed from `logical_and` and `logical_not` productions. The `errors` and `typerep` attributes are defined explicitly so that an error message can be reported on the code written by the programmer. The value of the `c` attribute is defined implicitly and retrieved from the forwards-to tree. Forwarding is used in the implementation of language extensions to define their translation to the host language. Forwarding suffices for translations that require only a local transformation.

The definitions of `typerep` are described below in Section 2.2. Productions

defining statements, declarations, and other expressions are what one might expect and are not shown. Also, several definitions that would have the expected value are elided with ellipses (...).

The production for functions calls completes Fig. 1. Its definition of `typerep` is not specified here, but is given in the `aspect` production with the same name in Fig. 3. Aspect productions allow attributes to be defined for concrete or abstract productions specified in different locations in the same file, different files, or even different modules.

2.1.2 Silver specifications for concrete syntax

Productions with the `concrete` modifier are used to generate input specifications for a parser generator. Different extensions to Silver integrate different parser and scanner generators into Silver. The original version of Silver generates input to Happy – a Yacc-like LALR(1) parser generator that creates parsers implemented in Haskell, the language in which Silver specifications are implemented. The current version of Silver generates input to Copper [48]. This is an LALR(1) parser and scanner generator that we have designed that uses *context-aware scanning*. In this approach, each time the parser calls the scanner to retrieve the next token, the parser passes to the scanner the set of terminal symbols that are allowed in the current parse state. This set is called the *valid lookahead set*. The scanner will only return tokens in this set and it is thus more discriminating than traditional disjoint scanners. With this approach parse table conflicts are far less likely when composing grammars. In fact, when the concrete syntax introduced by an extension satisfies a set of reasonable restrictions we can guarantee that there will be no conflicts in the composed grammar [40].

The full range of attribute declaration and definition capabilities in Silver can be used on concrete productions just as they are used on abstract productions. Thus, on languages in which the concrete syntax is straightforward one may decide to perform semantic analysis on the concrete syntax tree. For more complex languages, one may separate the concrete and abstract syntax so that the only attributes on the concrete productions are used to construct the AST over which the attributes that implement the semantic analysis (such as type checking) are evaluated.

The separation of concrete and abstract syntax in SimpleC is a bit contrived but its specification is shown in Fig. 2. Nonterminals are defined just as in the abstract syntax with the exception that `Prog_c` is marked as the *start* nonterminal. The concrete nonterminals and productions are named by convention with a `_c` suffix to indicate that they are part of the concrete syntax.

Next are the declarations of some terminal symbols. `Id_t` and the regular ex-

```
grammar edu:umn:cs:melt:simplec;
start nonterminal Prog_c ;
nonterminal FunDcls_c, FunDcl_c, Dcls_c, Dcl_c,
            Stmts_c, Stmt_c, Type_c, Expr_c ;

terminal IntLit_t  /[0-9]+/ ;
terminal Id_t      /[a-zA-Z][a-zA-Z0-9\_]*/ ;
terminal NotOp '!'  precedence = 12;
terminal AndOp '&&' precedence = 10, association = none ;
terminal OrOp  '||' precedence =  8, association = none ;

synthesized attribute ast_Stmt_c    :: Stmt occurs on Stmt_c ;
synthesized attribute ast_Stmts_c   :: Stmt occurs on Stmts_c ;
synthesized attribute ast_Expr_c    :: Expr occurs on Expr_c ;

concrete production program_c  p::Prog_c ::= fds::FunDcls_c
 { p.ast_Prog_c = program(fds.ast_FunDcls_c) ; }
concrete production stmts_cons_c
 ss::Stmts_c ::= s::Stmt_c stail::Stmts_c
 { ss.ast_Stmts_c = stmt_seq(s.ast_Stmt_c, stail.ast_Stmts_c) ; }
concrete production stmts_one_c  ss::Stmts_c ::= s::Stmt_c
 { ss.ast_Stmts_c = s.ast_Stmt_c ; }
concrete production while_c
 w::Stmt_c ::= 'while' '(' c::Expr_c ')' s::Stmt_c
 { w.ast_Stmt_c = while(c.ast_Expr_c, s.ast_Stmt_c) ; }
concrete production assign_c
a::Stmt_c ::= id::Id_t '=' e::Expr_c ';'
 { a.ast_Stmt_c = assign(id,e.ast_Expr_c) ; }
concrete production logical_and_c
 e::Expr_c ::= l::Expr_c '&&' r::Expr_c
 { a.ast_Expr_c = logical_and(l.ast_Expr_C, r.ast_Expr_c) ; }
concrete production idref_c  e::Expr_c ::= id::Id_t
 { e.ast_Expr_c = idref(id); }
```

Fig. 2. Concrete syntax specification for SimpleC.

pression (denoted `/`*regex*`/`) are used by the generated scanner to identify identifiers. Keyword and punctuation terminal symbols, like `AndOp`, that match a fixed string (denoted `'`*fixed lexeme*`'`) instead of a regular expression can be specified by their fixed string directly in productions, as in the production `logical_and_c`. Traditional specification of operator precedence and associativity are also supported.

Silver is a strongly-typed language and thus several synthesized attributes of the appropriate types are defined in order to compute the abstract syntax tree from the concrete syntax tree. The first one in Fig. 2 specifies that the attribute `ast_Stmt_c` is used to generate ASTs of type `Stmt` for concrete nonterminals of type `Stmt_c`. Definitions of these attributes are provided on the concrete

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

autocopy attribute env :: [ Binding ] ;
attribute env occurs on Prog, Dcls, Dcl, Stmt, Expr, Type ;
nonterminal Binding with typerep ;
synthesized attribute name :: String occurs on Binding ;
synthesized attribute typerep :: TRep occurs on Expr ;
nonterminal TRep ;

abstract production funcType     ft::TRep ::= i::TRep o::TRep { }
abstract production booleanType  bt::TRep ::=                 { }
abstract production intType      it::TRep ::=                 { }
abstract production arrayType    at::TRep ::= elem::TRep      { }
abstract production errorType    et::TRep ::=                 { }

synthesized attribute errors :: [String] collect with ++ ;
attribute errors occurs on Prog, Dcls, Dcl, Stmt, Expr, Type ;

aspect production func_call  e::Expr ::= f::Id_t arg::Expr
{ e.c = f.lexeme ++ "(" ++ arg.c ++ ")" ;
  e.errors <- case ftype of
                funcType(i,o) =>
                         (if   equal_types(i,arg.typerep)
                          then [ :: String ]
                          else ["Error: incorrect input type."])
              | _ => ["Error: functional type required."] end ;
  e.typerep = case ftype of
                funcType(i,o) => o
              | _ => errorType() end ;
  local attribute ftype :: TRep ;
  ftype = lookup(e.env,f.lexeme) ; }
```

Fig. 3. Specifications for type checking SimpleC.

productions and construct the AST. Many of the specifications are omitted but can be inferred. This specification is rather verbose when compared to tools like Yacc and in Section 3.3 we describe an extension to Silver that provides a much more concise and readable specification mechanism for concrete syntax.

2.2 Full Silver: core Silver with language extensions

The Silver constructs used in Figs. 1 and 2 use primarily constructs from the core Silver language. The definitions of attributes `errors` and `env` in Fig. 1 and the specifications in Fig. 3 make use of Silver features that were added as extensions to the core Silver language. These features include type-safe polymorphic lists, pattern matching on syntax trees, collection attributes,

and some extensions that allow one to conveniently define the attribution (`occurs on`) relation. In Section 3.3 we describe an extension for the concise specification of concrete syntax and in Section 3.4 we describe an extension for the specification of data-flow analysis for imperative programs.

The inherited environment (symbol-table) attribute `env` defined in Fig. 3 uses two of the extensions to Silver. First, it is an `autocopy` attribute and thus if no explicit definition for `env` is given in a production, then one is automatically generated that copies the value of `env` from the left-hand side nonterminal node to its appropriate children. Second, its type uses the type-safe polymorphic list extension to specify that `env` is a list of `Binding` values. The simple `Binding` nonterminal declaration uses the `with`-clause extension to indicate that the `typerep` attribute decorates `Binding`. The attribute `name` also decorates `Binding` nonterminals. These are used to bind names of program variables to *type-representation* trees of type `TRep`.

The attribute `typerep` is a higher-order attribute. It holds a tree whose root is a nonterminal of type `TRep`. The type of an `Expr` is represented by these trees. These trees are constructed by the productions `funcType`, `booleanType`, `intType`, and `errorType` that follow. These productions are used to define the higher-order attribute `typerep` on expressions and bindings. In Fig. 1 the abstract production `booleanType` is used to indicate that logical expressions have boolean type.

Collection attributes in Silver are similar to those defined by Boyland [5] with the exception that assignments to collection attributes cannot be made via reference attributes to reference tree nodes decorated by such attributes. Collection attributes are associated with an associative operator used to fold together contributions to the attribute. Collection attributes are declared using the `collect with` clause that specifies the collection operator. The Silver collection assignment operator `:=` (which differs from the standard definition operator `=`) is used in several productions to define the attribute's initial value. Aspect productions may use the collection contribution operator `<-` to fold additional values into the attribute. A fold operation of type $((a \times a \rightarrow a) \times a \times [a]) \rightarrow a$ uses the operator, initial value, and list of contributed values assigned in different aspects to compute the final value of the attribute. A collection attribute with operator $\oplus$, initial value v_i and values assigned in aspects $v_1, v_2, ..., v_n$ has the final value of $v_i \oplus v_1 \oplus v_2 \oplus, ..., \oplus v_n$. Although the operator does not have to be commutative, the order in which aspect-contributed values are combined is not specifiable in Silver and thus this order must not matter. In Fig. 3 the `errors` attribute of type `[String]` is collected by the list concatenation operator `++`. On the while-loop production `while` in Fig. 1 the initial value of this attribute is the concatenation of the errors found on its two children. In the `func_call` production in Fig. 1 the initial value is the errors on the argument `arg`. This is combined with the

errors defined in the aspect production in Fig. 3. This contribution (`<-`) to `errors` uses pattern matching to inspect the production used to construct the `typerep` tree `ftype`. This tree is the type of the identifier `f` as found in the environment `e.env` using the `lookup` function.

The production `program` in Fig. 1 defines the attribute `env` to be the empty list of bindings. Silver does not have ML-style type-inference and it is not part of the polymorphic list extension. Thus, empty list expressions explicitly specify the type of the list elements.

Pattern matching is a mechanism for data structure decomposition used in combination with algebraic datatype definitions and found in several languages including ML and Haskell. The matching of a value to a pattern is performed by matching the value constructors of each variant of the datatype to that of the value, and recursively matching the arguments to the value constructors of each. In Silver, and in AGs in general, nonterminals correspond to algebraic types and productions correspond to value constructors for variants of the datatype. The production signature is a device similar to the ML datatype definition, by which both auxiliary data structures and the abstract syntax of the object language can be defined in a uniform way. When nonterminals and productions are used for constructing syntax trees for programs there are rather few occasions in which one needs to know what the variant (production) of a tree is and thus attribute grammar systems do not typically have constructs for doing so. However, when AG constructs are used to define data-structures that are used more generally, decomposition often becomes quite useful and sometimes necessary.

This can be seen in Fig. 3 in the aspect production `func_call`. The type for SimpleC expressions is represented by the datatype (nonterminal) `TRep` and each abstract production with a `TRep` left-hand side nonterminal defines a *variant* of the datatype. To perform type checking on function calls, the input type and output type would be extracted from the constructed functional type; on array access expressions, the array's component type must be extracted. Without pattern matching, synthesized attributes would need to be defined for these component types. But this cannot be done in a type-safe manner since on any `TRep` production most such attributes would not be properly defined; e.g., we would either not define a `funcOutputType` attribute on `arrayType` productions, or define it with some sort of error value. Pattern matching provides a type-safe solution and is used in the aspect production `func_call` in Fig. 3 to specify that the type of a function call is the output type of the type of the function being called. In the case that the type of the identifier `f` is not a function, an error is generated. A *local attribute* (`local attribute`) is used to hold the type of the function. Because the attribute definitions and local attribute declarations are not ordered it is not uncommon to write the definition of local attributes near the end of this collection, after it has been

```
grammar silver:core ;
start nonterminal File ;

nonterminal AGDcls, AGDcl;
concrete production fileRoot
 f::File ::= g::GrammarSpec i::Imports dcls::AGDcls
 { f.haskell = ...;
   dcls.env = i.defs ++ dcls.defs; }

synthesized attribute haskell :: String ;

concrete production ntDcl d::AGDcl ::= 'nonterminal' nt::Id ';'
 { ... }
concrete production occursDcl
 d::AGDcl ::= 'attribute' attr::Id 'occurs' 'on' nt::Id ';'
 { ... }
concrete production agDclCons  ds::AGDcls ::= d::AGDcl dtail::AGDcls
 { ... }
abstract production agDclSeq  ds::AGDcl ::= d1::AGDcl d2::AGDcl
 { ... }
```

Fig. 4. Sample specifications of the `silver:core` language.

used.

3 Implementing Silver and its language extensions

This section shows how some of the Silver extensions described in Section 2 are implemented as language extensions and composed with core Silver. The full-featured version of Silver used to specify SimpleC is constructed from the host language core Silver and the extensions described above and in Sections 3.3 and 3.4. This core host language is implemented as an attribute grammar in the module `silver:core`. The extensions to Silver are implemented as attribute grammar fragments that extend `silver:core`. Silver is implemented in Silver via bootstrapping. For example, we built collection attributes as an extension to Silver and used it to enable other language extensions, such as the pattern matching extension shown below. A few declarations in the specification of core Silver are shown in Fig. 4. It declares nonterminals for a Silver file and attribute grammar declaration(s) (`AGDcl`, `AGDcls`) that are used in the abstract production declarations for nonterminals (`ntDcl`) and occurs-on declarations (`occursDcl`). The grammar is implemented by a translation to Haskell specified by the `haskell` attribute defined on core Silver productions. Some additional Silver specifications are provided later in this section and used to describe the implementation of the language extensions.

```
grammar silver:exts:convenience ;
concrete production withDcl
d::AGDcl ::= n::'nonterminal' nt::Id 'with' attr::Id s::';'
{ d.errors = ... check that nt and attr are declared ...
  forwards to agDclSeq ( ntDcl(n,nt,s),
              occursDcl('attribute',attr,'occurs', 'on', nt, s) ) }
```

Fig. 5. Partial Silver specification for the simplified *with*-clause extension.

3.1 The With-*Clause*

A nonterminal declaration using the *with*-clause in Silver additionally specifies that the listed attributes occur on the declared nonterminal. It is a simple extension that requires only a local transformation to translate into core Silver. The declaration `nonterminal Binding with typerep;` in Fig. 3 translates to

```
nonterminal Binding; attribute typerep occurs on Binding;
```

The implementation of a simplified version of the *with*-clause extension (that specifies only one nonterminal and one attribute) is shown in Fig. 5 as part of the `silver:exts:convenience` module. The production `withDcl` explicitly defines an attribute `errors` so that error messages can be issued in terms of the specification written by the developer (not the generated specification). Other attribute values are implicitly defined by and obtained from its forwards-to tree, the syntax of its semantically equivalent series of nonterminal and `occurs on` declarations.

3.2 Pattern Matching

In order to implement pattern matching as a modular language extension to core Silver, both local and additive global transformations are required in translating pattern matching constructs into core Silver. Note that only a small part of the core Silver and pattern matching specifications is shown in an effort to provide a relatively detailed description of one aspect of the implementation as opposed to a broad but shallow overview. Consider the `case` expression that defines `typerep` in the aspect production `func_call` in Fig. 3. A local transformation, implemented via forwarding, translates this construct to the core Silver nested `if-then-else` expression shown in Fig. 6, the details of which are described below.

```
if   ftype.prodName == "funcType"
then cast(TRep,get_nth(ftype.childList,1))
else if true then errorType()
             else error("No matching pattern for case expression");
```

Fig. 6. Result of local transformation of pattern matching `case` to core Silver.

```
grammar edu:umn:cs:melt:simplec ;
synthesized attribute prodName :: String ;
synthesized attribute childList :: [ AnyType ] ;
attribute prodName, childList occurs on TRep ;

abstract production funcType  ft::TRep ::= i::TRep o::TRep
{ ft.prodname = "funcType" ;
  ft.children = [ cast(AnyType,i), cast(AnyType,o) ] ; ... }
abstract prod boolType    bt::TRep ::=
{ bt.prodname = "boolType" ;
  bt.children = [ :: AnyType ] ; ... }

aspect prod funcCall
{ e.typerep =  ... see Fig. 6 .. ;   ... }
```

Fig. 7. Result of local and global transformation mapping the SimpleC grammar to core Silver.

Global transformations add the declarations, occurs-on declarations, and definitions of the attributes `prodName` and `childList` used in the translation of a pattern matching `case` construct, like the one in Fig. 6. These are added on a global scale to the object grammar. Part of the transformed SimpleC grammar is shown in Fig. 7. The local transformations, as we have seen in the SimpleC `logical_or` and Silver `withDcl` constructs, are easily implemented via forwarding. This is briefly covered below before the discussion of the implementation of the global transformations which is the main topic of this section.

The local transformation is implemented using forwarding in the same manner as with the simplified `with` declaration shown above. The productions defining case expressions use a higher-order attribute (not shown) to construct the nested *if* expression that the case expression forwards to. This expression uses two attributes; `prodName` of type `String` that holds the name of the production used to construct the tree, and `childList`, a list of `AnyType` values that are the nonterminal trees and terminals that were the right-hand side arguments to the production. In Fig. 6, we test the `prodName` attribute to determine which pattern matches `ftype`. If it was constructed by the production `funcType` then the `get_nth` function extracts the proper list element which is cast back to the proper type (`TRep`). This makes use of a type-unsafe `AnyType` type in core Silver that is useful in language extensions such as this one. (Section 3.5 discusses how type-safety is restored in the extended version of Silver used for specifying languages other than Silver and used in our specifications of SimpleC and Java 1.4.) The type `AnyType` wraps terminal, nonterminal and primitive types in a single type and the `cast` operator is used to wrap or unwrap these values. A run-time error is raised if these are used incorrectly.

We focus on the global transformation that adds attribute definitions for

```
grammar silver:core ;
concrete production prodDcl
 p::AGDcl ::= 'abstract' 'production' n::Id sig::Signature
              b::ProdStmts
 { production attribute moreStmts :: ProdStmts
                        collect with prodStmtsSeq ;
   moreStmts := prodStmtsEmpty();
   forwards to prodDcl_expanded (n, sig,
                                 prodStmtsSeq(b, moreStmts) ) ; }
abstract production prodDcl_expanded
 p::AGDcl ::= n::Id sig::Signature b::ProdStmts
 { p.haskell = ...;   ... }

abstract production prodStmtsSeq
 p::ProdStmts ::= p1::ProdStmts p2::ProdStmts {...}
abstract prod prodStmtsEmpty p::ProdStmts ::=   {...}
```

Fig. 8. Building extensibility into production declarations.

`prodName` and `childList` to productions in the object grammar. The transformations that add the declarations and occurs-on declarations are done in a similar manner. These transformations are additive and do not impede or conflict with other additive global transformations of the kind described in Section 1.1 since it only adds declarations and attribute definitions to productions. (It is the responsibility of the developer of the global transformation to ensure that it can in fact be composed with other extensions. Name clashes are the primary concern but these are easily handled as the implementation of Silver uses fully-qualified names based on unique module names.)

Silver is designed for certain types of extensibility in order to support global transformations that add new constructs into the object grammar (*e.g.* SimpleC). The extension points which allow this are implemented by a pair of productions, one that collects the new constructs, and one used in constructing the translation to core Silver. For production declarations, these two productions (`prodDcl` and `prodDcl_expanded`) are shown in Fig. 8. The concrete production `prodDcl` is used by Silver's parser to construct the original tree of the object grammar. (In Silver, unlike SimpleC, we perform semantic analysis on the concrete syntax tree.) The tree of the `func_call` production in Fig. 1, for example, is constructed using this production. The production `prodDcl` has a collection attribute `moreStmts` that collects all the new attribute definitions that are to be added by global transformations, such as those defined in the pattern matching extension. The `production` modifier on this attribute declaration indicates that it is visible on all aspect productions on `prodDcl`. As we will see in Fig. 9, the grammar defining pattern matching has a `prodDcl` aspect production that contributes to this collection attribute the definitions of `prodName` and `childList`. The statements collected in `moreStmts` are folded together using the sequence production `prodStmtsSeq`. These and the existing

```
grammar silver:exts:patternmatching ;
aspect production prodDcl
p::AGDcl ::= 'abstract' 'production' n::Id sig::Signature
             b::ProdStmts
{  moreStmts <- "sig.lhs.name.prodName =  n.lexeme;" ;
   moreStmts <- "sig.lhs.name.childList =  sig.rhs.childList;" ; }
```

Fig. 9. Adding object language declarations for pattern matching.

statements in the body of the production in the original AST (named `b`) are combined to form the set of production statements that appear in the translation to core Silver. The second production in the pair, `prodDcl_expanded`, uses these as the body of the production-declaration tree that the "collecting" production `prodDcl` forwards to. For the `funcType` production of SimpleC in Fig. 4, this forwarded-to tree forms its translation to core Silver and is the result of the global transformations. It is shown in Fig. 7.

Fig. 9 shows a small part of the `silver:exts:patternmatching` grammar module that specifies the global transformation that adds definitions of the new attributes to the existing object grammar productions. This is accomplished by an aspect production on `prodDcl` that adds the new attribute definitions to the `moreStmts` attribute using the collection operator `<-`. We give a stylized specification of the actual productions; in between the double quotes (*"..."*) elements in typewriter font depict the concrete syntax of the attribute definition statements being added to the collection attribute and elements in italics are instantiated with values from the production. The actual specification builds Silver constructs explicitly using the productions that define the Silver language. The composition of the core Silver grammar and the pattern matching grammar has the effect of adding attribute definitions for attributes `prodName` and `childList` to each production declaration of an object language specification. Note that in defining contributions to collection attributes like `moreStmts` the developer must take care not to introduce any new attribute dependencies that might cause a circular attribute dependency.

3.3 Concise Concrete Syntax

The concrete syntax specifications in Fig. 2 are quite verbose. The reason for this is that concrete productions allow one to define attributes in the same way as one does on abstract productions. This general expressiveness is quite useful in language specifications in which semantic analysis (that is, attribute evaluation) is performed on the concrete syntax and no separate abstract syntax tree is constructed. The specifications for Silver itself make use of this. In other languages it is often more useful to define a separate abstract syntax for semantic analysis. For such languages, the expressiveness of Silver concrete productions is not needed and a much more concise mechanism for

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

start Prog_c ast Prog ;
FunDcls_c ast Dcls ;  FunDcl_c  ast Dcl ;
Dcls_c    ast Dcls ;  Dcl_c     ast Dcl ;
Stmts_c   ast Stmt ;  Stmt_c    ast Stmt ;
Type_c    ast Type ;  Expr_c    ast Expr ;

Prog_c    ::= FunDcls_c           { program($1) }
FunDcls_c ::= FunDcl_c FunDcls_c  { dcls_cons($1, $2) }
            | FunDcl_c            { dcls_one($1) }
Stmts_c   ::= Stmt_c Stmts_c      { stmt_seq($1, $2) }
            | Stmt_c              { $1 }
Stmt_c    ::= 'while' '(' Expr_c ')' Stmt_c  { while($3,$5) }
            | Id_t '=' Expr_c ';'            { assign($1,$3) }
            | '{' Dcls_c Stmts_c '}'         { block($2,$3) }
Dcls_c    ::= Dcl_c Dcls_c     { dcls_cons($1,$2) }
            |                  { dcls_none() }
Dcl_c     ::= Type_c Id_t ';'  { var_dcl($1,$2) }
Type_c    ::= Int_kwd          { int_type() }
Expr_c    ::= IntLit_t         { intconst($1) }
            | Id_t             { idref($1) }
            | Expr_c '&&' Expr_c { logical_and($1,$3) }
            | Expr_c '||' Expr_c { logical_or($1,$3) }
            | '!' Expr_c         { logical_not($2) }
```

Fig. 10. Concise concrete syntax specification for SimpleC.

writing concrete syntax specifications can be introduced.

In this section we describe just such an extension. It provides notations similar to those found in Yacc [26] and other parser generator tools that supports a single un-named synthesized attribute typically used to construct the abstract syntax tree of the program being parsed. The specification of a portion of the concrete syntax of SimpleC, but not the specification of terminal symbols, can be seen in Fig. 10.

Nonterminals that are used in the concrete syntax grammar specify the nonterminal (or more generally, Silver type expression) used in the abstract grammar to which they map. For example, the concrete syntax nonterminal `Expr_c` maps to the nonterminal `Expr` used in the abstract grammar in Fig. 1. This mapping is indicated by the `ast` clause on the abbreviated nonterminal declarations. Terminal symbols are used in both the concrete and abstract syntax and thus the identity mapping is used for these.

The concrete productions are written in more concise notation since productions do not need to be named and since names are not associated with the nonterminals and terminals in the production. The single synthesized attribute

that is computed on these production is defined by the expression in brackets following the production. Here, the familiar `$`i notation is used to indicate the synthesized attribute computed on the i^{th} symbol on the right-hand side of the production.

Using forwarding the constructs in Fig. 10 translate to core Silver language constructs similar to those in Fig. 2. The synthesized attributes (*e.g.* `ast_Expr_c`) for constructing the AST are automatically generated by the extension during translation of the new notation to core Silver constructs.

3.4 Data-flow analysis in Silver

This section describes a language extension to Silver that does not add general purpose features, such as the pattern matching constructs in Section 3.2, but instead adds constructs that are domain-specific. In this case, for the domain of analysis of imperative programs.

When writing the specifications of imperative languages such as SimpleC, it would be useful to be able to specify certain well-known and verified data-flow analyses (such as detection of "dead" code) in a high-level and declarative way, without having to implement them directly. One such declarative framework is that of temporal logic. Data-flow properties may be specified as temporal logic formulas which are model-checked on the control flow graphs (CFGs) to obtain the results of the analysis [42,39]. For example the following optimization, with a trigger condition expressed as a formula in the temporal logic CTL-FV [31], performs dead-code elimination:

$$s : x := expr \Rightarrow \texttt{skip} \quad if \quad s \models AX\ A\ [\neg use(x)\ W\ (def(x)\ \&\&\ \neg use(x))\ ]$$

The optimization removes the assignment $x := expr$ if the side-condition holds on the corresponding node s in the CFG of the program. The formula is true if there is no path from s to any future state where the variable x is used and has not been redefined. More precisely, the formula is true if on all next (`AX`) states from s, on all paths (`A`), x is either never used, or it is not used until it is defined (again), with the new definition not using the old value. This optimization is only valid if expressions do not have side-effects. We could add an analysis to determine which functions are side-effect-free and incorporate that into this optimization but we leave it out here to simplify the presentation.

In this section, we describe an extension to core Silver that allows the compiler writer to transcribe well-known and verified [2] data-flow analyses such

[2] Previous work [32] has shown how to prove the correctness of transformations such as the one mentioned here.

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

cfg nodes Stmt, Expr;
cfg attributes def, uses;
synthesized attribute entry:: CFG_Node occurs on Stmt;
inherited   attribute succ :: CFG_Node occurs on Stmt;

aspect production assign  s::Stmt ::= x::Id expr::Expr
{ s.entry = cfg s [s.succ];
  s.def = x.lexeme;   s.uses = expr.uses;  }

aspect production stmt_seq  s::Stmt ::= s1::Stmt s2::Stmt
{ s.entry = s1.entry; s1.succ = s2.entry; s2.succ = s.succ; }

aspect production while  w::Stmt ::= cond::Expr body::Stmt
{ w.entry = cfg cond [body.entry, w.succ];
  body.succ = w.entry ;  }
```

Fig. 11. Constructing CFGs for programs in SimpleC

as the one above directly into their Silver language specification. The results of the analysis can be used in an integrated way with other analyses such as type checking, which are typically done on the abstract syntax trees of programs. There is thus a single declarative framework for doing multiple kinds of analyses.

The extension adds constructs that allow the compiler writer to specify how the CFG is to be created from the AST. The nodes of the CFG are labelled with values obtained from attributes on the corresponding nodes of the AST. The decorating attributes depend on the formula to be model checked. The syntax of core Silver is extended to include temporal logic formulas and constructs that evaluate the formulas on nodes in the CFG. The results of evaluating the formulas are obtained by external system calls to model checkers (NuSMV [8] in the case of CTL) and are returned and available to the rest of the attribute grammar.

Fig. 11 shows part of the specification in this extended version of Silver that creates control flow graphs for programs in SimpleC. The `cfg nodes` construct is used to specify which type of AST nodes may be used to construct the CFG (in this case `Stmt` and `Expr`). Each `Stmt` node in the AST is associated with a subgraph in the final control flow graph. On assignments, this subgraph is just the single CFG node created for the assignment. On while loops, this subgraph consists of the CFG node created for the condition (`Expr`) and the subgraph corresponding to the body of the loop. To compute the CFG and link the various subgraphs together, two attributes, `entry` and `succ`, are defined. The synthesized attribute `entry` is defined on a statement to point to the entry

node of the subgraph of the CFG for that statement. The inherited attribute `succ` on a statement stores the (pointer to the) node which will follow its CFG subgraph. New CFG nodes are created using the `cfg` keyword with the syntax

`cfg` ⟨*corresponding AST node*⟩ ⟨*successor nodes*⟩

For example, the entry node of the subgraph of the while loop is a new node constructed from the conditional expression `cond`. This node has as its successors the entry node of the subgraph corresponding to the body of the while loop, and the successor node of the while loop itself. Finally, `body`'s CFG subgraph is succeeded by `cond`'s CFG node, since control flows from the body of the loop back to the conditional expression.

The specification also uses the `cfg attributes` construct to specify which attributes decorating the AST nodes will also be used in the data-flow analysis and must thus label the nodes in the CFG. The values of these attributes are used to generate the actual CFG (model) that is processed by the external model checker. Here the values of the string-valued attribute `def`, (which contains the name of the variable (if any) assigned at a particular node) and the attribute `uses` of type `[String]` (containing the names of variables used at a particular node) are specified as labelling the nodes of the CFG. The values of these attributes are used in the the temporal logic formula that is the trigger of the optimization that performs dead code elimination. The fact that `def` and `uses` have been declared as attributes that decorate the CFG nodes implies that they must decorate all AST nonterminals declared as CFG nodes (in this case `Stmt` and `Expr`).

Fig. 12 shows code that includes the dead-code elimination optimization mentioned at the beginning of this section. The optimization is specified in the `assign` aspect in the definition of the attribute `opt_stmt` which stores the "optimized statement". It is defined to be either the skip statement or the original assignment statement, based on whether the boolean-valued model checking expression evaluates to true. The syntax of the model checking expression is $M, s \models f$, where M, s and f are the model, state and formula respectively. Here, the state corresponding to the assignment node in the NuSMV model is model checked against the CTL formula which is true if the assignment is dead. It may be noted that the optimization described at the beginning of this section can be transcribed nearly word-for-word into the Silver code. The formula in the `assign` production is instantiated for each assignment in the program with the lexeme of the identifier being assigned to. Thus for the assignment `a = 0;`, the formula

```
AX A [!(a in uses) W (def == a && !(a in uses)) ]
```

is generated and translated into its NuSMV representation and sent to the model checker along with the NuSMV representation of the model CFG. These

```
autocopy attribute smv_model :: SMV_Model occurs on Stmt ;

aspect production func_decl f::Dcl ::= f::Id args::Params body::Stmt
{ body.smv_model = smvmodel body.entry
                    [def ranges over f.all_vars,
                     uses ranges over powerset of f.all_vars];
}

aspect production assign  s::Stmt ::= x::Id expr::Expr
{ s.opt_stmt = if s.smv_model, s.entry |=
                   AX A [!(x.lexeme in uses) W
                         (def == x.lexeme && !(x.lexeme in uses)) ]
               then skip ()   else s;
}
```

Fig. 12. An optimization that performs dead-code elimination triggered by a side–condition specified in CTL.

new constructs are translated using forwarding into pure non-extended core Silver code containing Silver system calls that call the model checker. This process is not described in detail here as it is similar to the process used by the pattern matching extension. Further details can be found in a previously published workshop paper [46].

The NuSMV model is created in an aspect to the `func_dcl` production (only intra-procedural data-flow analysis is performed here) using the `smvmodel` keyword, which takes the initial node of the CFG and the ranges of the attributes that label the CFG nodes. The initial node of the CFG will be the initial node of the body of the function. The ranges need to be given here as NuSMV requires that model specifications specify the (finite) domains of all state variables. Here the attribute `all_vars` stores the list of all variables in the function. The inherited attribute `smv_model` passes the NuSMV model to all parts of the syntax tree so that it can be used to perform data-flow analysis wherever needed.

While this section presented only one example of a logic (CTL) and its model checker (NuSMV), extensions using other logics and model checkers can also be easily written. In fact, they can be combined to allow the compiler writer to use different logics to specify different data-flow properties within the same grammar. This approach thus provides a flexible and high-level way to specify and use the results of data-flow analyses in a language specification, one closely integrated closely with the rest of the (syntax-tree driven) attribute grammar specification.

```
grammar silver:full;

import silver:core with syntax hiding cast_cs anyType_cs;

import silver:exts:convenience with syntax;
import silver:exts:collection with syntax;
import silver:exts:patternmatching with syntax;
import silver:exts:list with syntax;
import silver:exts:concise_concrete_syntax with syntax;

abstract production main   m::Main ::= args::String
{
 forwards to silver_driver(args, parse);
}
```

Fig. 13. Composition of grammars to create `silver:full`.

3.5 Composing core Silver and its extensions to create full featured Silver

To build a full-featured extended version of Silver that has the convenience extensions such as the with-clause and autocopy inherited attributes, collection attributes, pattern matching, and type-safe polymorphic lists we compose the core Silver language and these extensions in the Silver specification in Fig. 13. This specification composes the attribute grammars that are imported and composes their concrete specifications (when imported with the `with syntax` clause). The semantics of import are as if the imported grammar (but not what it imports) was textually included directly in the importing file. The `hiding` clause is a mechanism for excluding certain items from being imported into a grammar specification. This is used above to ensure that `silver:full` is type-safe by not importing into the grammar the concrete syntax of the type-unsafe constructs `AnyType` and `cast`.

The `main` production plays a role that is similar to `main` in C and takes the command line arguments as its `String`-type parameter. This production forwards to the Silver driver production that controls compilation of Silver grammars. It passes this its arguments and the parser that recognizes the language composed of the concrete syntax specifications that are imported.

The specifications shown throughout this section have by necessity been rather brief and we have omitted some non-critical aspects of Silver, its extensions, and their implementation. The complete specifications for Silver and it extensions can be found at `www.melt.cs.umn.edu`.

4 Related Work

There are many ways to implement DSLs [37] and Silver is not the only declarative system that supports modular language design. Modular language definition and extensibility have received much attention from the AG community [19,27,20], to mention just a few. Some systems are guided by functional programming ideas and use, in essence, higher-order functions as attributes in their quest for modular specifications [17,1]. Others are inspired by the object-oriented paradigm and employ inheritance to achieve a separation of concerns [24,36]. Well-defined circular attributes [16] and generic attributes [38] have also been studied.

Well-developed AG systems such as LRC [30], JastAdd [14], Eli [22], and UUAG [2] support a wide range of useful attribute grammar features such as JastAdd's reference attributes for retrieving attribute values from remote nodes in the tree and Eli's *constituents* for easily collecting information from nodes in a production's sub-trees. However, these systems do not support forwarding and thus the modularity and ease-of-composition of language features specified as AG fragments are often achieved by writing attribute definitions that "glue" new fragments into the host language AG. To the best of our knowledge, JastAdd is the only AG tool that allows for the implicit specification of semantics by translation to a host or core language. This is done by the application of (destructive) rewrite rules. But attributes values are returned from the rewritten trees only, and thus one cannot both implicitly (via forwarding) and explicitly (via attribute definitions) specify the relevant semantics of new language constructs. Note that local attributes can be computed during rewriting to drive the rewriting process. These rewrites are not restricted to ensure composability and thus can be used in a wider variety of applications.

The general purpose features of pattern matching and polymorphic lists added to Silver are not strictly necessary in Turing complete AG systems with higher-order attributes. They are also not found in AG systems that have a "back-door" to the implementation language. This approach is taken by JastAdd (implemented in Java), Eli [22] (implemented in C), and others. But this can lead to an AG system that has a "split-personality" in that part of the problem is solved as an AG and part in the implementation language. Furthermore, one should avoid side-effecting computations in the implementation language as these can be difficult to reason about. One also needs to be careful that the computations written in the implementation language do not introduce additional attribute dependencies that are not visible to an analysis, like circularity detection, as these may invalidate the results of the analysis. For general purpose tasks, the back-door approach is not necessarily a bad idea. But it provides no support for adding additional domain-specific constructs,

such as those for pattern matching or collection attributes.

More generally, there are other approaches for specifying languages and language extensions. Embedded domain-specific languages [25,34] and macro systems (whether traditional syntactic [7,33], hygienic [29,9] or programmable [51]) allow the addition of new constructs to a language but they lack an effective way to specify semantic analysis and report domain-specific error messages. Some modern macro systems [4,3] however do a better job at this. Other well-developed declarative systems based on term rewriting include ASF+SDF [43] which has been used in many applications. Another is Visser's JavaBorg[6] that allows one to extend a host language by adding concrete syntax for objects. This system is based on StrategoXT [49] and uses strategies and term rewriting to process programs. Specifying semantic analyses, like error checking, as rewrite rules are less straightforward than it is using attributes and it is not clear that different extensions can be as easily combined.

Intentional Programming originated in Microsoft Research and proposed forwarding in a non-attribute grammar setting. The original work and some more recent work [41] uses a highly-developed structure editor for program input since traditional LR parsing of extensible languages was not seen as viable. But as mentioned, our context-aware scanning approach [48], which is used in an LR-parser, reduces the likelihood of parse table conflicts. This makes it possible to "certify" language extensions to provide a guarantee that when several independently certified extensions are later composed, there will be on conflicts in the parse table for the composed concrete syntax grammar. This is done by imposing a modest set of restrictions on concrete syntax that can be added to the host language [40]. We believe that this makes the use of LR-parsers viable for extensible languages.

Parsing expression grammars [18,23] have recently gained some attention and they are closed under composition, but the order in which they are composed can have an effect on the language that is recognized. This is another parsing technique that may be used in extensible language systems. Similarly, other work [12] based on meta-object protocols for language extension uses the GUI facilities in Eclipse for program input.

5 Discussion

We have introduced Silver, a full-featured extensible attribute grammar specification language that has been used to define implementations of and extensions to Java 1.4 [47], a subset of Lustre [21], and Silver itself. Different full-featured versions of Silver are implemented as the composition of a core Silver language and various general purpose and domain-specific language ex-

tensions. Silver supports the specification of composable local and additive global transformations. Higher-order attributes, forwarding, and collection attributes have not previously been available in a single AG system and were initially developed by different research groups. While none of these features are themselves new, a framework in which one can easily combine different general purpose and domain-specific features is. It is their combination and means of application that form the mechanisms that implement our extensible language methodology. These general-purpose and domain-specific additions to core Silver reflect the need for language evolution. In Silver, the evolution is achieved by adding these new features as modular extensions to the host language, core Silver.

Silver's ability to specify both local and additive global transformations is quite useful in implementing expressive language features. Forwarding provides a significant degree of flexibility in determining which semantics and translations (also implemented as a set of attributes) are defined explicitly and which are defined implicitly. A macro-like extension would define no synthesized attributes and get all semantics defined by the forwards-to construct. Forwarding and collection attributes allow the host language designer to build extension points that language extensions use to implement the additive global transformations that are often needed for more powerful language extensions.

Although we have demonstrated how several interesting enhancements to Silver can be implemented as language extensions, not all changes can be so easily accomplished. Consider adding ML-style type-inference as a language extension. While it is relatively straightforward to define *new* attributes that implement type-inference, integrating this into an existing typed language requires changes to how existing constructs know what their type is; that is, what attribute, an existing one, or a new one, contains the type representation for a construct. Silver does not have ML-style type-inference and it is not part of the polymorphic list extension. Thus, the empty list expression explicitly specifies the type of the list elements.

Circular attributes are another attribute grammar feature that is difficult to add as a language extension. Eva Magnusson's Ph.D. thesis [35] provides interesting examples of the use of circular attributes - one computes the *nullable*, *first*, and *follow* sets for a context-free grammar used in parser construction. Circular attributes can be seen as changing the fix-point operation used in the attribute evaluator. This is a significant change to the semantics of the core language and is probably best done in the underlying AG evaluator and not as a language extension.

6 Acknowledgements

This work is partially funded by the McKnight Foundation and the National Science Foundation via NSF CAREER Award #0347860 and NSF CCF Award #0429640.

References

[1] S. R. Adams. *Modular Grammars for Programming Language Prototyping*. PhD thesis, University of Southampton, Department of Elec. and Comp. Sci., UK, 1993.

[2] A. Baars, D. Swierstra, and A. Loh. Utrecht University AG system manual. http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem.

[3] J. Baker and W. Hsieh. Maya: Multiple-dispatch syntax extension in Java. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–281. ACM, June 2002.

[4] D. Batory, D. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–53. IEEE, 2–5 1998.

[5] J. Boyland. Remote attribute grammars. *J. ACM*, 52(4):627–687, 2005.

[6] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proc. of OOPSLA '04 Conf.*, pages 365–383, 2004.

[7] T.E. Jr. Cheatham. The introduction of definitional facilities into higher level programming languages. In *AFIPS (Fall Joint Computer Conference, 29)*, pages 623–637, 1966.

[8] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Proc. Computer Aided Verification*, pages 495–499, 1999.

[9] W. Clinger. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–162. ACM Press, 1991.

[10] K. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[11] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[12] A. D. Eisenberg and G. Kiczales. Expressive programs through presentation extension. In *AOSD '07: Proc. of the 6th Intl. Conf. on Aspect-oriented Software Development*, pages 73–84, 2007.

[13] T. Ekman. *Extensible Compiler Construction*. PhD thesis, Lund University, Lund, Sweden, 2006.

[14] T. Ekman and G. Hedin. Rewritable reference attributed grammars. In *Euro. Conf. on Object-Oriented Prog., ECOOP'04*, volume 3086 of *LNCS*, pages 144–169, June 2004.

[15] T. Ekman and G .Hedin. The JastAdd extensible Java compiler. In *Proc. Conf. on Object oriented programming systems and applications (OOPSLA)*, pages 1–18. ACM, 2007.

[16] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. *ACM SIGPLAN Notices*, 21(7), 1986.

[17] R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars. In *19th ACM Symp. on Prin. of Programming Languages (POPL)*, pages 223–234, 1992.

[18] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proc. of Symp. on Principles of Programming Languages (POPL)*, pages 111–122. ACM, 2004.

[19] H. Ganzinger. Increasing modularity and language-independency in automatically generated compilers. *Science of Computer Programing*, 3(3):223–278, 1983.

[20] H. Ganzinger and R. Giegerich. Attribute coupled grammars. *SIGPLAN Notices*, 19:157–170, 1984.

[21] J. Gao, M. Heimdahl, and E. Van Wyk. Flexible and extensible notations for modeling languages. In *Fundamental Approaches to Software Engineering, FASE 2007*, volume 4422 of *LNCS*, pages 102–116. Springer-Verlag, March 2007.

[22] R. Gray, S. Levi, V. Heuring, A. Sloane, and W. Waite. Eli: a complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, 1992.

[23] R. Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–51, New York, NY, USA, 2006. ACM Press.

[24] G. Hedin. An object-oriented notation for attribute grammars. In *Proceedings of the European Conference on Object-Oriented Programming, ECOOP'89*. Cambridge University Press, 1989.

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

[26] S.C Johnson. Yacc - yet another compiler compiler. Technical Report 32, Bell Laboratories, July 1975.

[27] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.

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

[29] E. Kohlbecker, D. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161. ACM Press, 1986.

[30] M. Kuiper and J. Saraiva LRC — a generator for incremental language-oriented tools. In *7th International Conference on Compiler Construction*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, 1998.

[31] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In *Proc. 10th Intl. Conf. on Compiler Construction*, volume 2027 of *LNCS*, pages 52–68. Springer-Verlag, 2001.

[32] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proc. 29th ACM Symposium on Principles of Programming Languages (POPL)*, pages 283–294. Association of Computing Machinery, 2002.

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

[34] D. Leijen and E. Meijer. Domain specific embedded compilers. *ACM Sigplan Notices, Papers of the 2nd USENIX Conference on Domain Specific Languages, 1999*, 35(1), January 2000.

[35] E. Magnusson. *Object-Oriented Declarative Program Analysis*. PhD thesis, Lund University, Department of Computer Science, Lund, Sweden, 2007.

[36] M. Mernik, M. Lenic, E. Avdicausevic, and V. Zumer. Multiple attribute grammar inheritance. *Informatica*, 24(3):319–328, 2000.

[37] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[38] J. Saraiva and D. Swierstra. Generic Attribute Grammars. In *2nd Workshop on Attribute Grammars and their Applications*, pages 185–204, 1999.

[39] D. Schmidt. Data flow analysis is model checking of abstract interpretations. In *ACM POPL Symp.*, pages 38–48, New York, NY, USA, 1998. ACM Press.

[40] A. Schwerdfeger and E. Van Wyk. Verifiable composition of deterministic grammars. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, June 2009.

[41] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. *SIGPLAN Not.*, 41(10):451–464, 2006.

[42] B. Steffen. Data flow analysis as model checking. In A.R. Meyer T. Ito, editor, *Theoretical Aspects of Computer Science (TACS'91)*, volume 526 of *LNCS*, pages 346–364. Springer-Verlag, September 1991.

[43] M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E.A. van der Meulen. Industrial applications of ASF+SDF. In *Algebraic Methodology and Software Technology (AMAST'96)*, pages 9–18. Springer-Verlag, 1996.

[44] E. Van Wyk, D. Bodin, and P. Huntington. Adding syntax and static analysis to libraries via extensible compilers and language extensions. In *Proc. of LCSD 2006, Library-Centric Software Design*, 2006.

[45] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th Intl. Conf. on Compiler Construction*, volume 2304 of *LNCS*, pages 128–142, 2002.

[46] E. Van Wyk and L. Krishnan. Using verified data-flow analysis-based optimizations in attribute grammars. In *Proc. Intl. Workshop on Compiler Optimization Meets Compiler Verification (COCV)*, April 2006.

[47] E. Van Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute grammar-based language extensions for Java. In *European Conf. on Object Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 575–599. Springer-Verlag, July 2007.

[48] E. Van Wyk and A. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *Intl. Conf. on Generative Programming and Component Engineering, (GPCE)*. ACM Press, October 2007.

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

[50] H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher-order attribute grammars. In *ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 131–145, 1990.

[51] D. Weise and R. Crew. Programmable syntax macros. *ACM SIGPLAN Notices*, 28(6), 1993.