Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 203 (2008) 103–116

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Silver: an Extensible Attribute Grammar System[1]

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 language, 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.

1 Introduction

Domain specific languages offer several significant advantages to their users over general purpose programming languages [5]. 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. But, domain specific languages have some disadvantages as well. Van Deursen et. al. [5, 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."

[1] This work is partially funded by NSF CAREER Award #0347860, NSF CCF Award #0429640, and the McKnight Foundation.

1571-0661 © 2008 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2008.03.047

- “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 [10] and there has been a steady stream of research in such systems since then, see [17,7,2] 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 1.4 [6,7] and Icon [9].

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 [14] 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 [16] and several modular language extensions. One embeds SQL into Java and performs static type-checking of the embedded SQL queries [13]. 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 [8].

In the early stages of this work, using a prototype implementation of Silver we found the challenges described by van Deursen et. al. [5] 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 [2] or auto-copy rules for inherited attributes to reduce boilerplate AG code. 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 [4, page 185] have noted that AGs can sometimes feel cumbersome and restrictive when compared to modern languages. Thus determining the scope of Silver, determining what domain specific and general purpose features should be implemented, and determining how to do it in a cost effective manner are all important considerations.

1.1 Extensible Languages

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 domain-adapted general purpose languages. We thus decided to implement Silver as an extensible language in order to mitigate some these challenges. Through a series of boot-strapping steps we were able to implement Silver as an AG specification written in Silver.

In this paradigm, languages are not treated as monolithic entities. 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 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 to semantically equivalent constructs in the host language. This provides an implicit specification of the semantics (that is attributes) of the extension. This is done via *forwarding* [14] which also allows *explicit* specification of semantics (attributes) at the extension language level.

Some features cannot be implemented by purely local transformations and require non-local transformations. Because we want language extensions to be composable, in the sense that the order in which extension language constructs are translated down to host language constructs should not matter, Silver does not support the sort of global transformations that cause radical rewrites of the of the original syntax tree.[2] 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 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

A core attribute grammar language serves as the host language for the full-featured version of Silver. In addition to the traditional constructs introduced by Knuth [10] the core Silver language includes *higher-order* attributes [17] that allow attributes to

[2] 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.

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 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. The core language also includes forwarding [14], a feature we introduced 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 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 features version. These include pattern matching on trees (by production), type-safe polymorphic lists, collection attributes [2], and convenience constructs such as auto-copy inherited attributes. Additional extensions provide constructs for building control flow graphs for imperative programs and performing dataflow analysis via model checking [15]. These extensions are discussed in Section 2.2 and 3. We will not provide formal definitions of attribute grammars [10], higher-order attributes [17], forwarding [14], or collection attributes [2] 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 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

A Silver grammar module contains AG declarations for non-terminals, terminals, productions, and attributes. Module names are based on Internet domain names, as in Java packages, 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. Each file begins with a declaration of the grammar name. Figure 1 contains the partial specification of SimpleC, its name given by 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. Order does not matter as declarations in a file are visible in the entire file and in other files in that same module. Line comments begin with "`--`".

```
grammar edu:umn:cs:melt:simplec;
start nonterminal Prog ;
nonterminal Dcl, Dcls, Type,
 Stmt, Stmts, Expr ;
terminal Id /[a-zA-Z][a-zA-Z0-9]+/;
terminal AndOp '&&' precedence = 10,
                association = none ;
terminal OrOp  '||' precedence =  8,
                association = none ;
terminal NotOp '!'  precedence = 12;

syn attr c :: String ;
attr c occurs on Prog, Dcl, Dcls,
         Type, Stmt, Stmts, Expr;

nonterminal TRep ;
syn attr typerep :: TRep ;
attr typerep occurs on Expr ;

concrete prod program
p::Prog ::= d::Dcls
{ p.c = "#include<stdio.h>" ++ d.c;
  p.errors := d.errors;
  d.env = [ :: Binding ];}

concrete prod logical_and
e::Expr ::= l::Expr '&&' r::Expr
{ e.c = ...;  e.errors := ... ;
  e.typerep = booleanType(); }
```

```
concrete prod logical_not
e::Expr ::= '!' ce::Expr
{ e.c = ...;  e.errors := ... ;
  e.typerep = booleanType(); }

abstract prod funcType
 ft::TRep ::= in::TRep out::TRep {...}
abstract prod booleanType
 bt::TRep ::=    {...}
abstract prod arrayType
 at::TRep ::= component::TRep {...}
abstract prod errorType
 et::TRep ::=    {...}

concrete prod funcCall
e::Expr ::= f::Id '(' arg::Expr ')'
{ e.c =...; e.errors := arg.errors;}

concrete prod logical_or
e::Expr ::= l::Expr '||' r::Expr
{ e.errors := ... ;
  e.typerep = booleanType();
  forwards to logical_not (
   logical_and (logical_not(l),
                logical_not(r)));
  --  l || r => ! (! l && ! r)
}
```

Fig. 1. A portion of the SimpleC specifications written in (primarily) core Silver.

Reading from the beginning of Figure 1 we see the declaration of non-terminal symbols `Prog` (the grammar start symbol), `Dcl` (declaration), `Dcls`, `Type` (type expressions), `Stmt` (statement), `Stmts`, and `Expr` (expression). Next is the declaration of the terminal symbol `Id` and the regular expression (denoted `/`*regex*`/`) 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`.

Next a *synthesized* attribute `c` of type `String` is declared. It contains the translation of SimpleC constructs to C and decorates the non-terminals specified in the `occurs on` clause. The attribute `typerep` is a higher-order attribute that holds trees whose root is a non-terminal of type `TRep`. The type of an `Expr` is represented by these trees.

Following are a few sample *production* declarations. Productions with the `concrete` modifier are used to generate the input specification to a parser generator. Different extensions to Silver integrate different parser and scanner generators into Silver. These extensions provide translations of concrete productions and terminal declarations to the input language of a parser/scanner generator. Productions marked as `abstract` or `aspect` are not used in the parser specification. The first production is named `program`, its left hand side non-terminal is `Prog` and is named `p`. The production's right hand side contains the `Dcls` non-terminal named `d`. Attribute definitions are given between the curly braces ({ and }). Here, the attribute `c` on `p` is defined as indicated. Definitions of other attributes that use features added as language extensions such lists (`[...]`) and collections (`:=`) are also shown but described below in Section 2.2. Attributes can be defined on concrete and abstract productions; for SimpleC we evaluate attributes on the concrete syntax tree since it is a simple language. 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 attributes are evaluated. Productions for conjunction and negation follow. These define the higher order attribute `typerep` to be the tree constructed by the abstract production `booleanType` to indicate that they are boolean expressions. Following are the abstract productions used to construct different type representations.

The concrete production for functions calls follows. Its definition of `typerep` is not specified here, but is given in the `aspect` production with the same name in Figure 2. 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. The pattern matching `case` expression is an extension to Silver and discussed below.

The `logical_or` production uses *forwarding* [14] to implement the local transformation that maps `l || r` to `!(!l && !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 a 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. 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 (...).

```
autocopy inh attr env::[ Binding ];

nonterminal Binding with typerep ;

syn attr name :: String
   occurs on Binding ;

syn attr errors :: [ String ]
 collect with ++ ;
attr errors occurs on Prog, Dcl,
 Dcls, Type, Stmt, Stmts, Expr ;
```

```
aspect prod funcCall
e::Expr ::= f::Id '(' arg::Expr ')'
{ e.typerep = case ftype of
     funcType(in, out) => out
   | _                 => errorType();
  e.errors <- case ftype of
     funcType(in, out) => [ :: Error ]
   | _  => [ "Error: " ++ f.pp ++
             " must be a function."];
  prod attr ftype :: TRep;
  ftype = ... lookup f in env ... ;  }
```

Fig. 2. A portion of the SimpleC specifications written in full Silver.

2.2 Full Silver: core Silver with language extensions

The definitions of attributes `errors` and `env` in Figure 1 and the specification in Figure 2 make use of Silver features that were added as extensions to the core Silver language. The inherited environment (symbol-table) attribute `env` defined in

Figure 2 uses two extensions. 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 is an extension that uses the `with` clause to indicate that the `typerep` attribute decorates `Binding`.

Collection attributes in Silver are similar to those defined by Boyland [2] and 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 attributes initial (or unit) value. Aspect productions may use the collection contribution operator `<-` to fold additional values into the attribute. A fold operation of type $((a \times a \to a) \times a \times [a]) \to a$ uses the operator, unit value, and list of contributed values assigned in different aspects to compute the final value of the attribute. A collection attribute with operator $\oplus$, unit value v_u and values assigned in aspects $v_1, v_2, ..., v_n$ has the final value of $v_u \oplus v_1 \oplus v_2 \oplus, ..., \oplus v_n$. 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 Figure 2 the `errors` attribute of type `[String]` is collected by the list concatenation operator `++`. In the `funcCall` production in Figure 1 the initial value is the errors on the argument `arg`. This is combined with the errors defined in the aspect in Figure 2.

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. In Silver, and in AGs in general, non-terminals correspond to algebraic types and productions correspond to value constructors for variants of the datatype. For example, Figure 2 shows a partial specification for type checking SimpleC function calls. The type for SimpleC expressions is represented by the datatype (non-terminal) `TRep` and each abstract production with a `TRep` left-hand side non-terminal 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 `funcCall` in Figure 2 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 *production attribute* (`prod attr`) is used to hold the type of the function. It is a "local" attribute visible only in the production body and in aspect productions of the defining production.

3 Implementing Silver and its language extensions

This section shows how two Silver extensions, the simple nonterminal-with declaration and pattern matching, 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. 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 collections 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 Figure 3. It declares non-terminals for a Silver file and attribute grammar declaration(s) (`AGDcl`, `AGDcls`) that are used in the abstract production declarations for non-terminals (`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.

```
grammar silver:core ;
start nonterminal File ;
nonterminal AGDcls, AGDcl;
abstract prod fileRoot
f::File ::= g::GrammarSpec
  i::Imports  dcls::AGDcls
{ f.haskell = ...;
  dcls.env = i.defs ++ dcls.defs; }
```

```
syn attr haskell :: String ;
abstract prod agDclSeq
 ds::AGDcl ::= d1::AGDcl d2::AGDcl
 { ... }
abstract prod ntDcl
 d::AGDcl ::= nt::Id { ... }
abstract prod occursDcl
 d::AGDcl ::= attr::Id nt::Id { ... }
```

Fig. 3. Sample specifications of the `silver:core` language.

3.1 The With*-Clause*

A non-terminal declaration using the *with*-clause in Silver additionally specifies that the listed attributes occur on the declared non-terminal. It is a simple extension that requires only a local transformation to translate into core Silver. The declaration `nonterminal Binding with typerep;` in Figure 2 translates to `nonterminal Binding; attr 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 Figure 4 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; other attribute values are implicitly defined by and obtained from its forwards-to tree, the abstract syntax of its semantically equivalent series of non-terminal and `occurs on` declarations.

```
grammar silver:exts:convenience ;
abstract prod withDcl  d::AGDcl ::= nt::Id  attr::Id
{ d.errors = ... check that nt and a are defined with correct type ...
  forwards to agDclSeq ( ntDcl(nt), occursDcl(attr, nt) ) }
```

Fig. 4. Partial Silver specification for the simplified *with*-clause extension.

3.2 Pattern Matching

In order to implement pattern matching as a composable, 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 are 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 `funcCall` in Figure 2. A local transformation, implemented via forwarding, translates this construct to the core Silver nested `if-then-else` expression shown in Figure 5, the details of which are described below. The global transformations add the declarations, occurs-on declarations, and

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

definitions of the attributes `prodName` and `childList` used in the translation of pattern matching `case` constructs, like the one in Figure 5. These are added on a global scale to the object grammar. Part of the transformed SimpleC grammar is shown in Figure 6. 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.

```
grammar edu:umn:cs:melt:simplec ;
...
syn attr prodName :: String ;  syn attr childList :: [ AnyType ] ;
attr prodName, childList occurs on TRep ;
...
abstract prod funcType    ft::TRep ::= in::TRep out::TRep
{ ft.prodname = "funcType" ;
  ft.children = [ cast(AnyType,in), cast(AnyType,out) ] ; ... }
abstract prod boolType    bt::TRep ::=
{ ft.prodname = "boolType" ;
  ft.children = [ :: AnyType ] ; ... }
...
aspect prod funcCall
{ e.typerep =  ... Fig. 5 .. ;   ... }
```

Fig. 6. Result of local and global transformation mapping the SimpleC grammar to core Silver.

The local transformation is implemented using forwarding in much 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 non-terminal trees and terminals that were the right-hand side arguments to the production. In Figure 5, 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 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 3.3 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, non-terminal and primitive types in a single type and the `cast` operator is use to wrap or unwrap these values.

We focus on the global transformation that adds attribute definitions for `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 transformation is additive and do not impede or conflict with other additive global transformations 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 of fully-qualified names based on unique module names.)

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

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

Fig. 7. Building extensibility into production declarations.

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 Figure 7. The abstract production `prodDcl` is used by Silver's parser to construct the original AST of the object grammar. The AST of the `funcType` production in Figure 1 is constructed using this 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. As we will see in Figure 8, 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 statements in the body of the production in the original AST (`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 Figure 3, this forwarded-to tree forms its translation to core Silver and is the result of the global transformations. It is shown in Figure 6).

Figure 8 shows a small part of the `silver:exts:patternmatching` grammar module that specifies the global transformation that adds definitions of the new attributes to 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 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 does need to take care to not introduce any new attribute dependencies that might cause a circular attribute dependency.

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

Fig. 8. Adding object language declarations for pattern matching.

3.3 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 auto-copy inherited attributes, collection attributes, pattern matching, and type-safe polymorphic lists we compose the core Silver language and these extensions in the following Silver 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;
abstract production main   m::Main ::= args::String
 { forwardsTo silver_driver(args, parse); }
```

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 extension (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 its extensions can be found at www.melt.cs.umn.edu.

4 Conclusion

4.1 Related Work

There are many ways to implement DSLs and Silver is not the only declarative system that supports modular language design. Well-developed AG systems such as LRC [11], JastAddII [7], and Eli [9] support a wide range of useful attribute grammar features such as JastAddII's reference attributes for retrieving attribute values from remote nodes in the tree and and Eli's *constituents* for easily collecting information from nodes in a productions sub-trees. However, these systems do not support forwarding and thus the modularity and ease-of-composition of language features specified as AG fragments is often achieved by writing attribute definitions that "glue" new fragments into the host language AG. To the best of our knowledge, JastAddII 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 JastAddII (implemented in Java), Eli [9] (implemented in C), and others. But this leads to AG systems that have a "split-personality" in that part of the problem is solved as an AG and part in the implementation language. Also, certain AG analyses, such as circularity analysis, are valid under the assumption that the unchecked implementation language code does not introduce cycles. 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. Macro systems (whether traditional syntactic, hygienic and programmable [18]) allow the addition of new constructs to a language but with the exception of a few modern macro systems, e.g. [1], they lack an effective way to specify semantic analysis and report domain specific error messages. Other well-developed declarative systems based on term rewriting include ASF+SDF [12] which has been used in many applications. Another is Visser's JavaBorg[3] that allows one to extend a host language by adding concrete syntax for objects. Specifying semantic analyses, like error checking, as rewrite rules is less straightforward than it is using attributes and it is not clear that different extensions can be as easily combined.

4.2 Discussion

We have introduced Silver, a full-featured extensible attribute grammar specification language that has been used to defined implementations of and extensions to Java 1.4 [16], a subset of Lustre [8], 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 extensions. It 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 is themselves new, a framework in which one can easily combine different general purpose and domain specific features is. 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 collection 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 allows the host language designer to build extension points which 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 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 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.

References

[1] Batory, D., D. Lofaso and Y. Smaragdakis, *JTS: tools for implementing domain-specific languages*, in: *Proc. 5th Intl. Conf. on Software Reuse* (1998).

[2] Boyland, J. T., *Remote attribute grammars*, J. ACM **52** (2005), pp. 627–687.

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

[4] Cooper, K. and L. Torczon, "Engineering a Compiler," Morgan Kaufmann, 2004.

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

[6] Ekman, T., "Extensible Compiler Construction," Ph.D. thesis, Lund University, Lund, Sweden (2006).

[7] Ekman, T. and G. Hedin, *Rewritable reference attributed grammars.*, in: *Euro. Conf. on Object-Oriented Prog., ECOOP'04*, LNCS **3086**, 2004, pp. 144–169.

[8] Gao, J., M. Heimdahl and E. Van Wyk, *Flexible and extensible notations for modeling languages*, in: *Fundamental Approaches to Software Engineering, FASE 2007*, LNCS **4422** (2007), pp. 102–116.

[9] Gray, R. W., S. P. Levi, V. P. Heuring, A. M. Sloane and W. M. Waite, *Eli: a complete, flexible compiler construction system*, CACM **35** (1992), pp. 121–130.

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

[11] Kuiper, M. and S. J., *LRC — a generator for incremental language-oriented tools*, in: *7th International Conference on Compiler Construction*, LNCS **1383** (1998), pp. 298–301.

[12] van den Brand, M., A. van Deursen, P. Klint, S. Klusener and E. van der Meulen, *Industrial applications of ASF+SDF*, in: *Algebraic Methodology and Software Technology (AMAST'96)* (1996), pp. 9–18.

[13] Van Wyk, E., 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.

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

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

[16] Van Wyk, E., L. Krishnan, A. Schwerdfeger and D. Bodin, *Attribute grammar-based language extensions for Java*, in: *European Conference on Object Oriented Programming (ECOOP)*, LNCS, 2007. To appear.

[17] Vogt, H., S. D. Swierstra and M. F. Kuiper, *Higher-order attribute grammars*, in: *ACM PLDI Conf.*, 1990, pp. 131–145.

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