A modular specification of Oberon0 using the Silver attribute grammar system

Ted Kaminski and Eric Van Wyk

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

Abstract

This paper describes an implementation of Oberon0 using the Silver attribute grammar system for the Tool Challenge at the 2011 International Workshop on Language Descriptions, Tools, and Applications. Silver was developed to study how independently-developed language extension specifications can be imported into a host language specification to define a new custom extended language. Thus it contains many features useful in modular language specification, such as forwarding, higher-order attributes, reference/remote attributes, and a simplified form of collection attributes. These are discussed in the context of the Oberon0 specification presented here.

Keywords: attribute grammars, extensible languages, language composition, Oberon

1. Introduction

In 2011, the International Workshop on Language Descriptions, Tools, and Applications (LDTA'11) held a Tool Challenge that invited developers and users of language processing tools, such as parser generators, attribute grammars, and term rewriting systems, to tackle a common problem in order to illustrate the commonalities and differences between various tools. The task was to implement a translator for Oberon0, a subset of Oberon used as an example language in Wirth's compiler construction textbook [1].

Since modular language design was a frequent theme of papers published at LDTA, the Oberon0 language was to be implemented as a sequence of language layers; starting with basic imperative constructs then adding new control flow constructs (for-loops and switch statements), procedures, and finally arrays and records. The tasks included parsing, pretty printing, name analysis, type checking, "de-sugaring" the new control flow constructs, lifting nested procedures to the top-level, and finally translating to C. These language levels and tasks are described in more detail in the preface [2] to the special issue in which this paper appears and later in this paper.

This paper describes an implementation of Oberon0 using Silver [3]. Silver is both a (meta) language for language specifications, as well as a tool that analyzes specifications and translates them into a Java-based implementation. A Silver module, called a grammar, can define the concrete syntax, abstract syntax (if one is used), the semantic analysis, and translation of a language or a language extension. The specification of lexical and concrete syntax of language constructs is based on context free grammars and regular expressions that are translated and passed to Copper, an integrated parser and context-aware scanner generator [4] coupled to Silver. Semantic analysis and translation tasks are specified as an attribute grammar and Silver supports many modern attribute grammar features, including higher-order attributes [5], reference attributes [6] (also called remote attributes [7]), and a simplified form of collection attributes [7]. Silver introduced *forwarding*, a mechanism that is useful for specifying composable language extensions, into an attribute grammar setting [8].

Silver also has many features found in modern functional programming languages. Silver is statically and strongly typed; its type system supports parametric polymorphism (as found in ML or Haskell) and a limited form of type inference. It also has higher-order functions and polymorphic lists, so many idioms from functional programming can be easily expressed. Many attribute grammar systems, including CoCoCo [9], JastAdd [10], Kiama [11, 12], and Silver, use a demand-driven mechanism similar to lazy evaluation to schedule, at attribute evaluation time, the evaluation of attribute equations. Both CoCoCo (an embedded DSL in Haskell) and Silver go a step further and also use lazy evaluation to evaluate the right hand side of attribute equations. It also has a module system and supports separate compilation. Silver can be seen as a safe, polymorphic, higher-order, lazy functional language with support for attribute evaluation [13].

Silver and Copper were developed to investigate a class of extensible languages that meet several criteria. First, extensions should be able to add new language constructs and new semantic analyses over them. In Silver, this amounts to supporting grammars (extensions) that add new nonterminals, new productions, and new attribute equations to those in the "host" language. Second, when extensions are composed with the host language, this should not require any modifications or additions to the host or extension specifications. These two criteria are commonly referred to as the *expression problem.*[1] Third, extensions may be developed independently [14]. Fourth, they can be composed automatically. Thus no additional specifications ("glue code") need be written to define the composition. As we will see later, forwarding provides a solution to this extension of the expression problem.

This composition of the host language and extension grammars is performed by Silver and Copper. Grammar composition is the straightforward set union of the different types of language specifications (sets of grammar symbols, productions, attribute equations, etc.), but the resulting specification may contain lexical ambiguities, define a grammar that is not LALR(1) (as required by Copper), or the attribute grammar may be missing required equations. To avoid this problem, Silver and Copper perform modular analyses on an extension and the host language[2] it extends that restrict extensions to prevent these problems from occurring. Silver performs a *modular well-definedness analysis* [15] that ensures that the composed attribute grammar is *well-defined*, meaning all attributes that may be evaluated have defining equations.[3] Copper has a *modular determinism analysis* [16] that ensures lexical ambiguities do not occur and that the composed grammar is LALR(1), something that only becomes practical when using a context-aware scanner as found in Copper. Together, these analyses let independent parties develop general-purpose or domain-specific language extensions that programmers, with no knowledge of the underlying specifications, can direct the tools to compose in order to create a unique language with the features they desire. This composition is guaranteed to "just work;" it results in attribute grammars that are complete (no missing equations) and scanner and parser specifications that have no ambiguities and can be parsed deterministically.

We have used Silver and Copper to develop a number of extensible lan-

[1]`http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`

[2]This host language may itself be an extended language created from a number of extensions. Extensions can extend other extensions directly. In both cases the analyses work the same way.

[3]When forwarding is used, as described in Section 4, not all attributes will be evaluated and thus this analysis does not simply check that equations exists for all attributes on all non-terminal symbols. Please see our previous papers for the details [8, 15].

guages host languages. These include the programming languages Java 1.4 [17] and more recently ANSI C (C11 standard) and modeling languages such as Promela [18], the specification language of the SPIN model checker, and Lustre [19], an asynchronous language used in safety critical systems. Silver[4] (implemented as a Silver specification) and Copper[5] are both distributed in source form under the Lesser GNU Public License (LGPL).

We encourage motivated readers to download the Silver specification of Oberon0[6] and explore the specifications while reading the examples in the paper as they have pointers to specific files.

2. The organization of the Oberon0 specification in Silver

To demonstrate Silver's support for highly modular language design, the Oberon0 specification is spread across many different Silver grammars. A grammar corresponds to a file system directory and is composed of the specifications in all Silver files (ending with a `.sv` extension) in that directory. (The scope of the specifications in one file includes all other files in that grammar.) We follow the example of package naming in Java and base grammar names (and directories) on an organization's Internet domain name to ensure that they are unique.

Named components (*e.g.* productions, attributes, non-terminals) have a *fully-qualified name* created from the unique grammar name and the name defined in the grammar. Thus undetected name-clashes are not possible in Silver. Grammars can import other grammars, optionally qualifying the imported names to avoid name clashes when two values with the same (not fully-qualified) name are to be used. Grammars can also export specifications from other grammars to create grammar specifications that are composed of multiple grammars. Thus a language designer has a high degree of flexibility in writing modular language specifications.

The language levels, named L1 – L4, and tasks, named T1 – T5, of the Tool Challenge are described in Table 1. Each language level includes the features from the levels below it. Task T3 depends on T2 and T5 depends on T4b.

[4]`http://melt.cs.umn.edu/silver`

[5]`http://melt.cs.umn.edu/copper`

[6]`http://melt.cs.umn.edu/oberon0`. A permanent archive of the specifications can also be found at `http://dx.doi.org/10.13020/D6H01F`.

L1	variable, constant, and type declarations, assignment statements, while-loops, if-statements, expressions with arithmetic, logical, and relational operators
L2	new control flow statements: for-loops and case statements
L3	procedures with by-value and by-reference parameters
L4	new data structures: arrays and records

T1	scanning, parsing, and pretty-printing	
T2	name analysis, reporting undeclared variables	
T3	type checking, reporting type errors	
T4	source to source transformation	
	T4a	translating L2 features to L1 features
	T4b	lifting nested procedures to the top level
T5	translation to C, after T4b	

Table 1: Language level and task descriptions.

The Silver Oberon0 specifications are organized across 5 top-level directories. The `core` grammar (with full name `edu:umn:cs:melt:Oberon0:core`) defines language L1 and tasks T1, T2, and T3. Its component sub-grammars further separate L1 into its concrete syntax (T1), abstract syntax (which defines T2), and type checking (T3). The `constructs` grammar has component grammars that define, respectively, control flow features (L2), data structures (L3) and procedures (L4). The `tasks` grammar and its components define the procedure-lifting transformation task (T4b) and the C code generation task (T5) for the various language levels.

The grammars in the `components` directory compose and export the relevant grammars from `core`, `constructs`, and `tasks` to define grammars for each language (L1 to L4) and each task (T1 to T5) and, for language components, define the appropriate parser. For example, Figure 1 shows the grammar defining language L4; it exports the appropriate grammars and also builds a parser from the concrete syntax specifications in them.

Finally, an `artifacts` directory contains grammars for the specific artifacts made up of various language levels and tasks as specified by the Tool Challenge. For example, as described in the preface to this special issue, artifact A1 is language L2 with tasks T1 and T2; artifact A2a adds L3 to A1 while A2b adds task T3 to A1. Artifact A3 is L3 and T3; artifact A4 is

```
grammar edu:umn:cs:melt:Oberon0:components:L4;

exports edu:umn:cs:melt:Oberon0:core;
exports edu:umn:cs:melt:Oberon0:constructs:controlFlow;
exports edu:umn:cs:melt:Oberon0:constructs:procedures;
exports edu:umn:cs:melt:Oberon0:constructs:dataStructures;

parser parse::Module_c {
  edu:umn:cs:melt:Oberon0:core;
  edu:umn:cs:melt:Oberon0:constructs:controlFlow;
  edu:umn:cs:melt:Oberon0:constructs:procedures;
  edu:umn:cs:melt:Oberon0:constructs:dataStructures;  }
```

Figure 1: Silver specification of the L4 component (file `Language.sv`).

```
grammar edu:umn:cs:melt:Oberon0:artifacts:A4;

imports edu:umn:cs:melt:Oberon0:components:L4 as L4;
imports edu:umn:cs:melt:Oberon0:components:T3;
imports edu:umn:cs:melt:Oberon0:components:T5;
imports edu:umn:cs:melt:Oberon0:core:driver only driver;

function main IOVal<Integer> ::= args::[String] mainIO::IO
{ return driver(args, L4:parse, mainIO); }
```

Figure 2: Silver specification of the artifact A4 (file `Artifact.sv`).

Lang. / Task	T1	T2	T3	T4a	T4b	T5	Total
L1	570	548	268	NA	349	263	1998
L2	138	125	39	34	0	0	336
L3	178	166	121	NA	184	94	743
L4	82	27	99	NA	28	30	266
Total	968	866	527	34	561	387	3343

Table 2: Lines of specifications for the tasks for different language levels.

L4 with all tasks. Silver `imports` and `exports` statements manage grammar composition. Figure 2 shows the specification of artifact A4; it imports language L4 and tasks T3 and T5. Since task T1 and T2 specifications are in the same grammars as the language level specifications they need not be imported here. Similarly, task T4 specifications need not be imported directly since T5 depends on them. The specification also imports a `driver` function that controls the execution of the translator using the parser defined in `L4`.[7] As can be seen, it is easy to create new artifacts with different combinations of language levels and tasks - it is essentially a matter of importing the appropriate component grammars.

Table 2 provides the number of lines used to specify the different tasks for different language levels. Note that these counts do not include the `components` and `artifacts` grammars as they primarily import and export statements that repackage the specifications in the `core`, `constructs`, and `tasks` grammars. Note, 285 lines contain a single left or right curly brace.

3. Scanning and parsing

Lexical and concrete syntax specifications in a grammar are processed by Copper, our integrated LALR(1) parser and *context-aware* scanner generator. Context-aware scanners [4] only scan for (and thus only return) terminals that are valid for the current context. In the case of LR parsers this context is the current LR parse state and the valid terminals are the ones that have a *shift*, *reduce*, or *accept*, but not *error*, as their parse table entry for the current state. Thus the principle of disambiguation by longest match is subordinate to the principle of disambiguation by context.

This type of scanning allows terminals that appear in different parsing contexts to have overlapping regular expressions. This is useful when parsing and scanning extensible languages in which different extensions may have terminals (*e.g.* new keywords) with overlapping regular expressions but they appear in different parsing contexts. More generally, terminal symbols need not be overloaded; instead different terminal symbols can be specified with overlapping regular expressions. This means that they are not overloaded in the parser and thus parse table conflicts are less likely to occur [4]. Since the concrete syntax of Oberon0 is so simple it is easily LALR(1) and there is no

[7]The `mainIO` is an artifact of Silver's rather archaic mechanism to manage side-effects in a pure lazy functional language.

need for context aware scanning. In our implementation no terminals with overlapping regular expressions are disambiguated by context.

Sample specifications - lexical syntax. Specifications for lexical syntax are familiar and consist primarily of terminal symbol declarations, which name the terminal symbol and provide the defining regular expression. In the Silver specification for the concrete syntax of L1[8], the `Id` terminal specifies variable names:

```
terminal Id /[A-Za-z][A-Za-z0-9]*/ ;
```

Terminals with regular expressions that match only one string, such as for keywords and operators, can specified as a single-quoted string. Productions in the concrete syntax can then use these quoted strings instead of the often less convenient terminal name. This can be seen for the assignment and addition operators below; the addition operator also shows the use of traditional precedence and associativity specifications:

```
terminal Assign_t ':=' ;
terminal Plus_t '+' precedence = 11, association = left;
```

There are two types of lexical disambiguation in Copper. The first specifies that a terminal t_1, for example a keyword, takes precedence over another t_2, for example an identifier, in all contexts in which t_2 is valid, even those in which t_1 is not valid. This is how keyword reservation can be specified. The second specifies a function over a set of terminals that chooses the one to return when a lexical ambiguity over those terminals occurs. This is used in a specification of ANSI C to disambiguate C variables from type names by keeping a parse-time list of type names defined by a `typedef`. The first form of disambiguation would be inappropriate for this because it would prefer the same terminal, either variable or type name, in all cases.

White-space and comment terminals are indicated by the `ignore` modifier that indicates a terminal is not to be passed on to the parser.

Sample specifications - context free concrete syntax. Specifications for concrete context free syntax also have a familiar structure; productions are written in BNF style and symbols may be preceded by a name which is used in

[8]Grammar `edu:umn:cs:melt:Oberon0:core:concreteSyntax`, file `Terminals.sv`.

```
grammar edu:umn:cs:melt:Oberon0:constructs:controlFlow:concreteSyntax;

concrete productions s::Stmt_c
 | 'FOR' id::Name_c ':=' lower::Expr_c 'TO' upper::Expr_c
     'DO' body::Stmts_c 'END'
   { s.ast = forStmt(id.ast, lower.ast, upper.ast, body.ast); }

 | 'FOR' id::Name_c ':=' lower::Expr_c 'TO' upper::Expr_c
     'BY' step::Expr_c 'DO' body::Stmts_c 'END'
   { s.ast = forStmtBy(id.ast, lower.ast, upper.ast, step.ast,
                       body.ast); }

synthesized attribute ast<a> :: a ;
attribute ast<Stmt> occurs on Stmt_c ;
attribute ast<Expr> occurs on Expr_c ;
```

Figure 3: Concrete syntax specifications for the for-loop (file `ForLoop.sv`).

assigning and accessing attribute values during attribute evaluation. Figure 3 shows the specification of the two for-loop constructs added in language L2. Since they are labeled as `concrete` they are used by Copper to generate a parser for the language. The attribute definitions, in curly braces, and the declarations for the attribute `ast` are discussed below.

4. Attribute Grammars, Attribute Evaluation

In attribute grammars [20], nonterminal symbols are associated with named values called *attributes*; nonterminal nodes in a syntax tree are decorated with these attributes. These attributes specify some semantic information such as a collection of semantic errors or the collection of in-scope variables. Equations associated with productions define the attribute values. *Synthesized* attributes propagate information up the syntax tree, and *inherited* attributes propagate contextual information down the tree. The types of attribute values in Silver include primitive values (*e.g* integers and strings), lists, syntax trees (called higher-order attributes [5]), or references to remote tree nodes (called reference or remote attributes [6, 7]).

Figure 3 shows the concrete productions for the for-loop and the attribute equations (written between curly braces). Nonterminal and terminal symbols are named in Silver productions, and attributes are referenced in equations using these names. The synthesized higher-order attribute `ast` is defined to be the abstract syntax tree for the construct that the attribute decorates. (Note that attributes can be defined on both concrete and abstract syntax.) For the first production, the equation defines the `ast` attribute of the left hand side nonterminal `s` using an abstract production (`forStmt`) and the `ast` attribute of its nonterminal children on the right hand side. Since punctuation, for example `:=`, and keywords do not appear in the AST, they are specified here using only the constant lexeme in single quotes that was given in the definition of that terminal symbol.

Silver supports parametric polymorphism as found in languages such as ML. The attribute `ast` is a polymorphic attribute that takes a type parameter named `a` (written between angle brackets). This specifies that `ast` has the type $\forall\alpha.\ \alpha$. This parametric type is instantiated by an `occurs on` declaration; on concrete nonterminals (`Expr_c` or `Stmt_c`) this attribute has corresponding type in the abstract syntax (`Expr` or `Stmt`).

Productions defining the abstract syntax, as shown in Figure 4, differ from concrete production in three ways: the `abstract` modifier is used, they have names unlike the anonymous productions shown in Figure 3, and they are not used to generate a parser. Otherwise there is no distinction between the two in Silver. Thus, one can define all the semantics of the language on the concrete syntax tree if desired. For Oberon0 we chose to use a separate abstract syntax, primarily for demonstration purposes.

The abstract production `forStmtBy` in Figure 4 provides equations defining (among others) the synthesized attributes `pp`, for pretty-printing the code, and `errors`, for collecting error messages. It also defines the inherited attribute `env` on the child nodes for propagating an environment (symbol table) down the syntax tree. This production uses forwarding to translate the for-loop to the semantically equivalent construct using a while-loop. Each of these is explained more fully in the following sections that describe the Silver implementation of the various language tasks for which these attributes and forwarding are used.

The `pp` attribute could have type `String` since these are easily concatenated to generate the pretty-printed version of the original program. However managing indentation and not generating too-long lines can be rather cumbersome. Thus Silver, like Kiama [11, 12] and Rascal [21], uses a pretty-printing library that handles such problems. In Silver we have implemented Swierstra and Chitil's [22] pretty-printer combinator library in which

```
grammar edu:umn:cs:melt:Oberon0:constructs:controlFlow:abstractSyntax ;
abstract production forStmtBy
s::Stmt ::= id::Name low::Expr up::Expr step::Expr body::Stmt
{ s.pp = concat( [
    text("FOR "), id.pp, text(" := "), lower.pp,
    text(" TO "), upper.pp, text(" DO"),
      nestlines(2, body.pp),
    text("DONE") ] );

  low.env = s.env;  up.env = s.env;  ...

  s.errors := low.errors ++ up.errors ++ step.errors ++
    body.errors ++
    case lookupDecl (id.name, s.env) of
    | nothing() -> [ err(s.location, id.name ++ " not declared") ]
    | just(dcl) -> [ ]
    end;

  forwards to seq(
    assign(idAccess(id), low),
    while(compareOperator(lExpr(idAccess(id)), up),
      seq( body,
           assign(idAccess(id), add(lExpr(idAccess(id)), step)))
    ) );
}
```

Figure 4: Abstract syntax and attribute specifications for the for-loop (file `ForLoop.sv`).

`Document` values are created (for example by the `text` function), combined (for example by the `concat` function), or appropriately indented. In Figure 4 the `nestlines` function creates a `Document` which indents, in this case, the lines of the for-loop body by two spaces. Additional features allow one to specify optional line breaks and other structure-defining features. When the final document is converted to a plain string, a maximum line length size is specified and a rather sophisticated analysis generates the string under these constraints. Libraries such as these make it much easier to generate nice looking textual representations of programs with a small amount of work.

5. Name analysis

Name analysis in the Silver implementation of Oberon0 proceeds in a standard way: the specifications[9] define an environment (symbol table) that is used to look up names, retrieve their declarations and included types, and report any re-declarations or undefined symbols; various helper functions are provided to assist in this. The environment is passed down the syntax tree as an inherited attribute `env`. This can be seen in the equations of the form `low.env = s.env;` in Figure 4. The `env` attribute is defined as an "autocopy" inherited attribute and thus if there is no equation defining `env` on a child then a copy equation such as those shown here are generated.

The function `lookupDecl`, with the declaration

```
function lookupDecl Maybe<Decorated Decl> ::= n::String e::Env,
```

takes a name to look up and an environment and returns either a `nothing()` value, indicating that the name was not found, or a `just(`d`)` value, in which d is a reference to the declaration (`Decl`) node in the tree where the name `n` was declared. The `Maybe` type is similar to the `Maybe` type in Haskell and the `option` type in ML. In Silver, this is a parameterized nonterminal type that has two abstract productions: the nullary `nothing` production and unary `just` production.

A use of this is seen in the definition of the `errors` attribute for `s` in Figure 4. (As discussed below, since `errors` is a *collection* attribute the `:=` symbol is used instead of the `=` symbol.) The type of `errors` is a list of messages, `[Message]`, and is defined here by combining the list of errors from the child nodes (using the list append operator `++`) and checking if the name `id` is defined. The case-expression matches on the result from a call to `lookupDecl` and creates the appropriate list of error messages. A result of `nothing()` indicates that the name was not declared and thus an error message is created. A result of `just(dcl)` indicates that `dcl` is a reference to the node that declared the name.

Decorated types. The type `Decorated Decl` is a *reference* [6] or *remote* [7] *attribute* and can be informally understood as a pointer or reference to a `Decl` node in the syntax tree. Whereas a higher-order attribute holds undecorated syntax tree values (essentially just terms in the language), a reference

[9]Grammar `edu:umn:cs:melt:Oberon0:core:abstractSyntax`, file `Env.sv`.

attribute is a reference to an existing, attributed, node in the tree. Attributes occurring on a `Decl` node can be retrieved via the reference attribute of type `Decorated Decl`, but they cannot be defined via the reference. Both JastAdd's reference attributes [6] and Boyland's remote attributes [7] support both accessing and defining collection attributes via a reference. Thus a production that uses an Oberon0 name calls `lookupDecl` to determine if the name is declared, and if so it can use the reference to access, for example, the `type` attribute on the `Decl`.

The environment. Oberon0 nests scopes in a standard manner; declarations in a scope hide declarations of the same name in any lexically enclosing scope. The type of attribute `env` is the non-terminal type `Env`. This higher-order attribute is a simple container that holds an attribute named `bindings` of type `[ TreeMap<String (Decorated Decl)> ]`. The bindings of names to declaration nodes in a single scope is represented by a map (`TreeMap`[10] is a red-black tree map) and the maps for all scopes are kept in a list, ordered from inner-most to outer-most. Note that type parameters, *e.g* `String` and `(Decorated Decl)` used to instantiate the polymorphic `TreeMap`, are separated by spaces and thus compound type expressions need to be written inside parenthesis. Here, the map (for a single scope) is from strings to decorated `Decl` nodes.

Specifications in a functional style. The function `lookupDecl`, see above, is used to retrieve the reference to a variable's declaration, if it exists, given the variables name and environment. The body of that function is

```
return foldr(orElse, nothing(), lookupInScopes(s, e.bindings));
```

Using lazy evaluation, this folds up the results of looking for the name `s` in all the scopes in the environment using a base-value of `nothing` and combining results with `orElse` to return the first result from `lookupInScopes` that is not a `nothing()`. Because Silver uses lazy evaluation this is efficient: the function will only actually do the lookup as far as is needed in the scope list. This style of programming is quite familiar to functional programmers using lazy languages and also demonstrates the usefulness of parametric polymorphism and higher-order functions in Silver.

[10]Grammar `silver:util:treemap`, file `TreeMap.sv`, in the Silver distribution.

The function `lookupInScopes`, where the type parameter `a` will be instantiated as `Decorated Decl` is shown below:

```
function lookupInScopes
[Maybe<a>] ::= s::String  ss::[TreeMap<String a>]
{return map(adapt, map(treeLookup(s,_), ss)); }
```

This function uses two applications of a traditional polymorphic `map` function. The `treeLookup` function takes a string and a tree-map and returns a list of all values (declarations) associated with the string in the tree-map. Mapping this function over the list of tree-maps `ss` returns a list of lists of declarations. The `adapt` function converts a list of values to a `nothing()` value if the list is empty or a `just` value containing the first element of the list. Thus the `map` using `adapt` returns a list of `Maybe<Decorated Decl>` values. Silver supports *partial application* of functions and that is used here where `treeLookup` is given only its first argument. The second is not given, as indicated by the underscore. The result of the partial application is a specialized lookup function for the name `s` that takes a tree as its only input. Again, lazy evaluation makes this efficient.

6. Type Checking

Specifications for type checking are spread across various grammars in the Silver Oberon0 specification and show how a new semantic analysis can be added to language constructs defined in other grammars. Silver's composition model allow a grammar to declare new attributes, indicate that they occur on nonterminals declared in another (imported) grammar, and define equations for these new attributes for existing productions in imported grammars.

The grammars that implement type checking, including the grammar `core:typechecking`[11], define a new synthesized attribute `type` that occurs on both `Decl` and `Expr`. On a declaration, this attribute is consulted via the reference attribute returned from `lookupDecl` to determine the type of a variable. On an expression, this attribute is computed to indicate its type. The type of the `type` attribute is `TypeRep`, a representation of types determined after type names have been resolved. (Alternatively, we could have

[11]We will omit the grammar prefix `edu:umn:cs:melt:Oberon0` to keep names short.

chosen `type` to have the type of `Decorated TypeExpr`; type expression constructs in declarations are represented by the type `TypeExpr` in the syntax tree.)

Aspect productions and collection attributes. We need to provide equations defining the `type` attribute for expressions (productions with `Expr` on their left hand side) and equations to report additional typing errors beyond what are reported when only name analysis is performed. Many of these productions are defined in the `core:abstractsyntax` grammar. *Aspect productions* specify new attribute equations for productions defined elsewhere, typically in other grammars. In this case aspect productions in `core:typechecking` associate attribute equations for the `type` attribute with the expression productions in `core:abstractsyntax`.

Figure 5 shows an aspect production for the production `intConst` that defines the `type` attribute on the production for integer constants to be a representation of the type integer. This uses the nullary production `integerType` with `TypeRep` on its left hand side. Similar aspects for other expression productions are defined in the `core:typechecking` grammar.

This example does not provide a complete picture of composition of language specifications in Silver and it does not provide a complete solution to the type checking problem. While we do not need to report any type checking error messages on the `intConst` production we certainly need to on others. *Collection attributes* in Silver allow aspect productions to add new error messages to the existing `errors` attribute defined in another grammar. Collection attributes are ordinary attributes that come with a *composition operator* that is used to combine all *contributions* to a collection attribute specified in different aspects for a production. For `errors`, this operator is the list append (`++`) operator as specified in the `with` clause of its declaration[12]:

```
synthesized attribute errors ::  [Message] with ++;
```

In Figure 4 the initial or base value of `errors` for a for-loop is specified using the `:=` operator, instead of the regular `=` operator. Besides collecting errors from the child nodes, the indexing variable is looked-up in the environment to ensure that it is declared. In Figure 6 additional error messages

[12]Grammar `silver:langutil`, file `Attributes.sv`, in the Silver distribution.

```
grammar edu:umn:cs:melt:Oberon0:core:typeChecking;
attribute type occurs on Expr ;
aspect production intConst  e::Expr ::= n::Integer
{  e.type = integerType();  }
```

Figure 5: Aspect production to type integer constants (file `Expr.sv`).

are contributed to the list of errors by definitions using the `<-` operator. These are added in an aspect production. Here, the indexing variable is looked-up in the environment again and if it is found and its type is not the integer type (as determined by the call to `check`) an additional error message is generated. In general, the final value of the collection attribute is equivalent to the expression `foldr(` *compositionOperator* `,` *baseDefinition* `,` `[` *contributions* `])`.

In Silver contributions to a collection attribute can only be made in a production or in aspect productions for it, not by following a reference or remote attribute which is supported by the type of collection attribute described by Boyland [7] and used in JastAdd. While the collection attributes in Silver are less expressive, this limitation simplifies that attribute evaluation algorithm and avoids the more complex algorithms that are otherwise required [23].

Collection attributes allow a language specification to provide specific ways in which language extensions can contribute to existing attributes. This is quite useful. In the case of `errors` the type checking extension can use the existing error-reporting facilities of the core language and does not need to define a new attributes, say `typeErrors`, to collect the typing error messages.

7. Source-to-source transformation

The effectiveness of *forwarding* [8] is illustrated in the specifications of the Challenge task (T2) that adds a for-loop and case-statement to the initial language, the code generation task (T5), and some additional simple syntax desugaring. Using forwarding, a production can create, at attribute evaluation time, a new syntax tree called the “forwards-to” tree. The original tree, the so-called “forwarding tree,” automatically forwards queries for attributes for which it does not have a defining equation to the forwards-to tree. The forwards-to tree is automatically provided with the same set of inherited attribute values as the forwarding tree. In this way, forwarding can be used

```
grammar edu:umn:cs:melt:Oberon0:constructs:controlFlow:typeChecking;

aspect production forStmtBy
s::Stmt ::= id::Name low::Expr up::Expr step::Expr body::Stmt
{ s.errors <-
    case lookupValue(id.name, s.env) of
       | nothing() -> [ ]
       | just(dcl) -> if check(dcl.type, integerType()) then []
                         else [ err(s.location, "... wrong type...")  ]
    end;  }
```

Figure 6: Aspect production for type checking a for-loop (file `ForLoop.sv`).

to provide default values to synthesized attributes. Another way to understand forwarding is that the forwards-to tree is simply a local higher-order (sometimes called *non-terminal*) attribute for which attribute equations are automatically generated that copy values from the forwarded-to tree to the node on which the forwarded-to tree was defined (the forwarding tree). The following examples show uses of forwarding for these tasks.

7.1. Desugaring

The specification of the for-loop production `forStmtBy` for task T2 in Figure 4 has a `forwards to` clause that defines this production's forwards-to tree as the expected sequence of an assignment statement and while-loop that the for-loop is translated to. This production and its aspect production in Figure 6 define `errors` so that messages are in terms of the constructs that the programmer wrote, not those that they translate to.

Note that there are no equations that define the `cTrans` attribute, the translation of the decorated construct to C. Instead the value for `cTrans` is defined by forwarding. When a for-loop tree is queried for this attribute value, it simply forwards that query to the forwards-to tree. Thus, the C translation of a for-loop is the C translation of the assignment/while-loop that it forwards to. A similar process is used for the case-statement. This is why the entries for L2/T4b and L2/T5 in Table 2 are both 0 lines of code.

Without forwarding or some other similar mechanism one must define *all* attributes for these simple control flow extensions; this can get tedious, even if they are simple copy equations. Forwarding is similar to traditional macros in that semantic analysis takes places on the forwarded-to tree /

expanded code. But with simple macros one cannot define any attributes and *all* semantic analysis is performed on the expanded (desugared) code. This is even less appealing since now extension-specific error checking is not possible. For example, one could not report an error that, say, indicates that only integer variables are allowed for indexing in for-loops since the semantic analysis takes place on the translated-to assignment/while-loop combination.

In addition to these T2 constructs, there are many language features that are simple syntactic sugar. The host language we have defined for Oberon0 has one example: `VAR` declarations can list multiple identifiers at once, with just one type. That is, both `VAR x,y : INTEGER` and `VAR x : INTEGER, y : INTEGER` have the same meaning. In the Silver implementation, the former declaration simply forwards to a sequence of the latter.[13]

7.2. Transformations to enable code generation

The procedure-lifting transformation (T4b), in the `tasks:lifting` grammar, is accomplished in Silver using higher-order attributes to create the "flattened" Oberon0 program in which all procedures are lifted to the top-level, as required by the target language C. We've implemented Johnsson's lambda-lifting transformation [24] to perform this task. It requires first renaming variables so that all are unique; it also solves a set of recursive equations used to compute the free-variables in recursive procedures that must now be passed in as arguments. Circular attributes could be used to compute the solutions to these equations, but since Silver lacks these the equations are passed up to the top of the AST where they are solved before the results are passed back down to where they can be used.

8. Code generation

The code generation process, in the `tasks:codegenC` grammar, is straightforward since the attributes for this are evaluated on the flattened version of the program generated using the lambda-lifting transformation. On the flattened tree all variable names are unique and all procedures have been lifted up to the top level. It is possible that the Oberon0 program has variables with the same name as C keywords. This problem is handled by using the renaming process in the lambda-lifting transformation. That process keeps

[13]Grammar `edu:umn:cs:melt:Oberon0:core:abstractSyntax`, `Declarations.sv`.

track of all names used in the program to detect variables which must be renamed. We initialize the collection of variable names to be all the C keywords, thus causing any Oberon0 variables with these names to be renamed. Thus the generation of C code boils down to little more than using a synthesized attribute of type `String` to un-parse the flattened Oberon0 AST using the concrete syntax of C.

9. Observations

The Oberon0 specification highlights some of the strengths and weaknesses of Silver. The module system and use of forwarding allows one to develop languages in a highly modular manner. On the other-hand, Silver is missing some helpful features such as a means for specifying abstract trees using concrete syntax and an easier-to-use mechanism for I/O.

Scanning and parsing. The simple concrete syntax of Oberon0 easily fits into the LALR(1) class and does not require the advantages of context-aware scanning. The only limitation in Copper exposed by Oberon0 is its inability to handle nested block comments since regular expression matching is used to match tokens. This gives rise to the only test failure in the Tool Challenge test suite.

Forwarding. Like many language specification techniques, forwarding lets one build up a language in layers, for example, the for-loop in L2 is built on top of the core L1 language. Forwarding allows us to avoid defining the semantics (attributes) for some new constructs that can be seen as syntactic sugar for those already existing in the language. This is seen in the fact that the for-loop gets it lifted-transformation semantics (T4b) and C translation (T5) from the L1 while-loop construct that it forwards/translates to. This is why there is a 0 in the L2/T4b and L2/T5 boxes in Table 2. But, it also allows us to define attributes for some semantics, for example type checking, when we want to do more than simply treat the new constructs as macros that translate down to other constructs without any semantic analysis. In this case we provide definitions for the type checking attributes so that better error messages can be provided to the programmer. This allows the language developer to build the language up in layers without having to treat non-core language features as simple macros or implement all of their semantics explicitly. Unlike many language specification techniques, forwarding does not lock the language engineer into defining either all or none of the semantics of a new language feature.

Modularity. We were also pleased that we could specify the different artifacts of the challenge in such a modular manner. The composition model of Silver is quite simple as it is just piece-wise set union of the grammar specifications: grammars are sets of productions, productions (concrete, abstract, and aspect) have sets of attribute definitions, and collection attributes are defined by sets of definitions spread across different grammars. This allows for the rather modular specifications as highlighted in Figures 1 and 2. This allows a language engineer to easily build languages from various components.

Modular Analysis. Attribute grammars can be used to solve the expression problem by adding new productions and new equations in aspect productions. Forwarding provides a solution to the stronger version of the expression problem described in Section 1. Extensions that add new attributes provide equations for the host language productions; productions added in independent extensions get values for the new attributes via forwarding to host language constructs. But we also want some assurance that the composition is well-defined and thus that no necessary attribute equations are missing.

An attribute grammar is *complete* if there are no missing equations. This is a simple static analysis that checks that for each production there is an equation defining each attribute that decorates the left hand side nonterminal and for each nonterminal on the right hand side there is an equation for each inherited attribute that decorates it. When forwarding is present, some of these equations are *not needed.* Consider the case in which a production computes all of its synthesized attributes via the tree it forwards to and does not access any synthesized attributes from its children: clearly, no equations for inherited attribute need to be present for those children. This leads to a notion of *effective completeness* [15]: for any syntax tree, during attribute evaluation an equation is present for any attribute value that is to be computed. A static effective completeness analysis is not as simple as the completeness analysis since it must take into account the flow of attribute values. This analysis uses the same attribute dependency information that is used in a circularity analysis [20].

Because we are interested in independently developed language extensions, we have developed a modular well-definedness analysis [15] that is performed on a single extension and the host language it extends. The composition of any set of extensions that individually pass this analysis will be effectively complete. This allows extension designers to check (and then modify as needed) their specifications so that there are no missing equations in the composed specification. This analysis puts some restrictions on extensions in order to provide these guarantees. One is that extension productions with a host language nonterminal on the left hand side must forward (directly or indirectly) to constructs in the host language. It also restricts equations so that new dependencies cannot be added to host language attributes on host language nonterminals. The host language determines what the dependencies on host language attribute and nonterminals are and extensions cannot extend these. For example, a host language attribute like `errors` depends on the inherited `env` attribute, but no others. An extension cannot contain an equation in which `errors` depends on additional inherited attributes. The details of this analysis are published in another paper [15].

This analysis is useful for independently developed language extensions, but it does not preclude an extension from depending on other extensions. All grammars that an extension grammar imports are seen as the so-called host language for this extension, even if some of those imported grammars are extensions to some original host language. In this case the imported extension grammars are not seen as being "independently developed" from the extension that imports them.

In the Oberon0 specification, the procedures introduced in L3 and data structures introduced in L4 do not pass the modular well-definedness analysis simply because there are no constructs in L1 into which these new constructs can be translated (via forwarding). This is required to solve the extended expression problem when another extension adds a new analysis to the host language since if L3 and L4 features do not forward to the host language constructs the composed compiler could not perform this analysis on those features.

In composing independently developed language extensions we must also be concerned about the concrete syntax specifications: is the grammar composed from the host language and the extensions ambiguous? Copper provides a modular determinism analysis for concrete syntax specification that provide similar guarantees to the modular well-definedness analysis. Any combination of extension concrete syntax specifications that pass the analysis, with respect to the host language, can be composed to form an LALR(1) grammar and a lexical syntax specification with no lexical ambiguities. The full description of this analysis can be found in an earlier paper [16].

Ideally, the for-loop and case-statements would be seen as composable language extensions, but the concrete syntax of both do not pass the modular determinism analysis since they both add new L2-defined terminals to the follow-sets [25] of L1-defined nonterminals. This can lead to parse-table conflicts in a composed language when two independently-developed extensions each add a new terminal to a host language nonterminal's follow-set that overlaps with no other terminals in the host language but that do overlap with each other. In this case, combining the host language with a single extension results in a conflict-free parse table, but the composition of both does not.

Both of the modular analyses work well when the host language is rich enough syntactically and semantically. Syntactically, this means that, for example, host nonterminals having large follow-sets. Semantically, this means that the language is expressive enough for extensions to have something to forward to. In our specification of ANSI C, we include the GCC statement-expression extension in our host language. This expression construct has the form: `({` $stmt_1$ `;` $stmt_2$ `;` ...`;` $stmt_n$ `;` $expr$ `; })`. It allows a sub-expression to contain a sequences of statements to be executed before the value of the sub-expression is computed as the value of the final expression. This is quite useful in adding expressive expression-level extensions to C since they can forward to this sort of construct. L1 as host language fails to meet these requirements and thus it is not surprising that these extensions do not pass the modular analyses.

10. Related Work

There are many different approaches to generating language implementations from declarative specifications in the literature. Here we focus on a comparison between Silver and other tools participating in the tool challenge and appearing in this special issue [2]. For a broader coverage of related work, please see other papers on Silver and Copper [8, 3, 13, 15, 4, 16, 26].

Safety. Silver and Copper emphasize safety in language specification, implementing a number of analyses designed to ensure that certain faults do not arise in the language implementations that are generated from declarative specifications. This is not unexpected since our focus is on composable language extensions. In this setting, a non-expert programmer will select a few independently-developed language extensions and thus the composition of these extensions with the host language must "just work."

Some of these analyses are useful in the Oberon0 specification. Copper requires grammars to be LALR(1), and thus non-ambiguous. Similarly, JastAdd's [10] use of Beaver and Simpl's use of ANTLR restrict the grammars to non-ambiguous classes. The other approaches prefer grammar formalisms that are not restricted in this way. Rascal uses generalized LL (GLL [27]) parsing and the OCaml implementation, Kiama, and CoCoCo use parser-combinator libraries. These all let one avoid the perceived restrictions of LALR(1) grammars at the cost of not checking at parser-specification time that the resulting parser will only return one syntax tree, or zero in case of a syntax error.

Silver's well-definedness analysis can be run over a monolithic (whole language) specification, as is done for Oberon0, ensuring that the attribute grammar is not missing any needed attribute-defining equations. CoCoCo gets a similar sort of safety by encoding the attribute completeness test in the Haskell type system - thus missing equations are reported, albeit as Haskell type errors. The other attribute grammar systems, JastAdd and Kiama, do not implement these sorts of analyses leaving the detection of missing equations to attribute evaluation time.

Ease of tool implementation. Building language processing tools can require a significant investment, but can also sometimes be carried out in a more lightweight manner which requires less investment of time and effort. The language processing tools described in this special issue range from high-investment "whole language" approaches to lightweight approaches.

The "whole-language" approach is taken by Silver and Rascal. These are systems in which the language specification is written in the Silver or Rascal *language* and that there is no commonly-used "backdoor" to the underlying implementation language. (Silver is translated to Java, but one doesn't write Java code in attribute equations.) This requires significant investment, but allows error messages to be generated for the Silver or Rascal specifications by the Silver or Rascal compiler. In Silver, it also makes it possible to implement Silver's type system and well-definedness analysis.

Kiama and CoCoCo are sophisticated embedded domain-specific languages (in Scala and Haskell, respectively) and rely on these underlying languages for at least some of their error checking and error reporting. This is similar for the OCaml implementation and Simpl which are even lighter weight approaches. While approach can reduce the investment in building the tool, it does come with the cost of errors that are sometimes not detected or are reported by the underlying languages type system which can make them less easy to understand. But the savings in implementation cost should not be underestimated. JastAdd specifications are a blend of these approaches, attribute and grammar specifications are written in the custom JastAdd language, while attribute equations are written in Java. Thus allowing the JastAdd system (which translates these specifications to Java) to report error messages on some components but also making it possible to get errors messages from the underlying Java compiler. A similar approach is taken by the AspectAG [28] system which can serve as a front end to the Haskell embedded domain-specific language CoCoCo.

Of course, no approach to tool implementation is clearly better than the others as demonstrated by the wide variety of approaches on display in the Tool Challenge and given the different goals of tool developers.

11. Conclusion

In this paper we have described the Silver implementation of Oberon0 developed for the LDTA'11 Tool Challenge. We have discussed aspects of the Silver specification of Oberon0 that highlight various characteristics of Silver.

To have a deeper understanding of Silver and its approach to language specification, interested readers are encouraged to download the Silver specification of Oberon0. By doing so, one can further explore the examples described above and other aspects of the implementation. The Tool Challenge test suite is included as are various scripts and instructions for building and running the Oberon0 artifacts. Additional documentation and discussion of the specification are also provided.

References

[1] N. Wirth, Compiler Construction, Addison-Wesley, 1996.

[2] M. van den Brand, Preface to the special issue on the LDTA'11 Tool Challenge, Science of Computer Programming. In press.

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

[4] E. Van Wyk, A. Schwerdfeger, Context-aware scanning for parsing extensible languages, in: Intl. Conf. on Generative Programming and Component Engineering, (GPCE), ACM, 2007, pp. 63–72.

[5] H. Vogt, S. D. Swierstra, M. F. Kuiper, Higher-order attribute grammars, in: Proc. of ACM Conf. on Programming Language Design and Implementation (PLDI), ACM, 1989, pp. 131–145.

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

[7] J. T. Boyland, Remote attribute grammars, J. ACM 52 (4) (2005) 627–687.

[8] E. Van Wyk, O. de Moor, K. Backhouse, P. Kwiatkowski, Forwarding in attribute grammars for modular language design, in: 11th Conf. on Compiler Construction (CC), Vol. 2304 of LNCS, Springer-Verlag, 2002, pp. 128–142.

[9] M. Viera, S. D. Swierstra, W. Swierstra, Attribute grammars fly first-class: How to do aspect oriented programming in Haskell, in: Proc. of 2009 International Conference on Functional Programming (ICFP'09), ACM, 2009, pp. 245–256.

[10] T. Ekman, G. Hedin, The JastAdd system - modular extensible compiler construction, Science of Computer Programming 69 (2007) 14–26.

[11] A. M. Sloane, Lightweight language processing in Kiama, in: Proc. of the 3rd summer school on Generative and transformational techniques in software engineering III (GTTSE 2009), Vol. 6491 of LNCS, Springer, 2011, pp. 408–425.

[12] A. M. Sloane, L. C. Kats, E. Visser, A pure embedding of attribute grammars, Science of Computer Programming 78 (10) (2013) 1752 – 1769.

[13] T. Kaminski, E. Van Wyk, Integrating attribute grammar and functional programming language features, in: Proc. of the 4th Intl. Conf. on Software Language Engineering (SLE 2011), Vol. 6940 of LNCS, Springer, 2011, pp. 263–282.

[14] M. Zenger, M. Odersky, Independently extensible solutions to the expression problem, in: Proc. of FOOL, Vol. 12, 2005.

[15] T. Kaminski, E. Van Wyk, Modular well-definedness analysis for attribute grammars, in: Proc. of Intl. Conf. on Software Language Engineering (SLE), Vol. 7745 of LNCS, Springer, 2012, pp. 352–371.

[16] A. Schwerdfeger, E. Van Wyk, Verifiable composition of deterministic grammars, in: Proc. of ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), ACM, 2009, pp. 199–210.

[17] E. Van Wyk, L. Krishnan, A. Schwerdfeger, D. Bodin, Attribute grammar-based language extensions for Java, in: Proc. of European Conf. on Object Oriented Prog. (ECOOP), Vol. 4609 of LNCS, Springer, 2007, pp. 575–599.

[18] Y. Mali, E. Van Wyk, Building extensible specifications and implementations of Promela with AbleP, in: Proc. of Intl. SPIN Workshop on Model Checking of Software, Vol. 6823 of LNCS, Springer, 2011, pp. 108–125.

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

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

[21] P. Klint, T. van der Storm, J. Vinju, Rascal: a domain specific language for source code analysis and manipulation, in: Proc. of Source Code Analysis and Manipulation (SCAM 2009), 2009.

[22] D. Swierstra, O. Chitil, Linear, bounded, functional pretty-printing, Journal of Functional Programming 19 (2009) 1–16.

[23] E. Magnusson, T. Ekman, G. Hedin, Demand-driven evaluation of collection attributes, Automated Software Engineering 16 (2) (2009) 291–322.

[24] T. Johnsson, Lambda lifting: transforming programs to recursive equations, in: Proc. of Functional Programming Languages and Computer Architecture, Vol. 201 of LNCS, Springer, 1985, pp. 190–203.

[25] A. Aho, R. Sethi, J. Ullman, Compilers – Principles, Techniques, and Tools, Addison-Wesley, Reading, MA, 1986.

[26] A. Schwerdfeger, E. Van Wyk, Verifiable parse table composition for deterministic parsing, in: 2nd International Conference on Software Language Engineering, Vol. 5969 of LNCS, Springer, 2010, pp. 184–203.

[27] E. Scott, A. Johnstone, GLL parsing, Electronic Notes in Theoretical Computer Science 235 (2010) 177–189.

[28] M. Viera, D. Swierstra, A. Middelkoop, UUAG meets AspectAG: How to make attribute grammars first-class, in: Proceedings of the 12^{th} Workshop on Language Descriptions, Tools, and Applications, LDTA '12, ACM, New York, NY, USA, 2012, pp. 6:1–6:8.