Sharing Trees and Contextual Information: Re-imagining Forwarding in Attribute Grammars

Lucas Kramer
krame505@umn.edu
University of Minnesota
USA

Eric Van Wyk
evw@umn.edu
University of Minnesota
USA

Abstract

It is not uncommon to design a programming language as a core language with additional features that define some semantic analyses, but delegate others to their translation to the core. Many analyses require contextual information, such as a typing environment. When this is the same for a term under a new feature and under that feature's core translation, then the term (and computations over it) can be shared, with context provided by the translation. This avoids redundant, and sometimes exponential computations. This paper brings sharing of terms and specification of context to forwarding, a language extensibility mechanism in attribute grammars. Here context is defined by equations for inherited attributes that provide (the same) values to shared trees. Applying these techniques to the ableC extensible C compiler replaced around 80% of the cases in which tree sharing was achieved by a crude mechanism that prevented sharing context specifications and limited language extensibility. It also replaced all cases in which this mechanism was used to avoid exponential computations and allowed the removal of many, now unneeded, inherited attribute equations.

CCS Concepts: • **Software and its engineering → Translator writing systems and compiler generators**.

Keywords: compilers, attribute grammars, modular and extensible languages, static analysis, well-definedness

ACM Reference Format:
Lucas Kramer and Eric Van Wyk. 2023. Sharing Trees and Contextual Information: Re-imagining Forwarding in Attribute Grammars. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623476.3623520

SLE '23, October 23–24, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0396-6/23/10…$15.00
https://doi.org/10.1145/3623476.3623520

```
1  for i in 0 to f(x) {  g(i);  }
```

```
1  { var i : int = 0;
2    var _v0 : int = f(x);
3    while i < _v0 {  g(i);  i := i + 1;  }  }
```

Figure 1. A for-loop example (top), introduced as a feature that translates to the code with a while-loop (bottom).

1 Introduction

Defining all semantic analyses, optimizations, and translations for all constructs in a full-featured programming languages can be a daunting task. One way to address this is to design the implementation with a smaller *core* language containing some collection of the essential language constructs and semantic analyses. Additional features are then layered on top of this core and provide a translation of the new feature into the core language, *e.g.* a for-loop with integer bounds may translate down to a while-loop in the core, as shown in Figure 1. This alleviates the need to specify certain semantic analyses or tasks, such as code generation, on non-core features by delegating them to the translation.

While other tasks, such as type checking, can also be delegated to the translation, this leads to reporting error messages on generated code instead of the code written by the programmer. If, *e.g.* the expression `f(x)` for the upper-bound on the for-loop in Figure 1 is not of type integer then an error about a type-mismatch on a variable declaration (line 2) may be reported and be nonsensical to the programmer. Thus, it is helpful to explicitly manage some aspects of compilation, such as type checking and error reporting, but dispatch other tasks, such as code generation, to the translation. Note that this requires managing the contextual information needed by sub-terms by passing this information down the syntax tree, *e.g.* an environment mapping variables to their types. In doing so, the language indicates that some semantic aspects of the non-core feature are *equivalent* to those of its translation but that some are not.

Not only does designing languages in this way save effort, it also leads to a more modular development as work on the core language (once it is established) can be isolated from work on non-core features and be carried out by different language developers. With proper tool support, modular

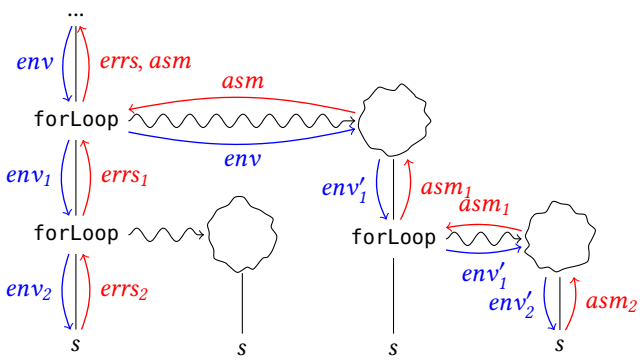

Figure 2. A portion of the decorated tree resulting from `for ... in ... { for ... in ... { s }}`. For clarity, the loop variable and bounds are not shown.

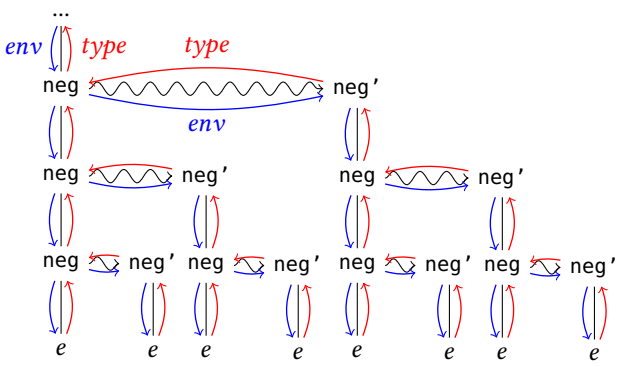

Figure 3. The resolution of operator overloading on a unary negation expression `~ (~ (~ e))` resulting in an exponential number of trees being created and traversed when computing `type`. `neg'` is the specialized implementation of `neg` for the type of *e*. Some labels are omitted for clarity.

design can enable a particular form of *extensible languages* in which new, often domain-specific, language constructs and static analyses can be developed independently, and composed together to form a language for problems that involve multiple domains. ExtendJ [3] and SugarJ [5] allow new features to be added to Java, and XOC [2] and ableC [11] allow new features to be added to C in a modular way.

Figure 2 shows aspects of the syntax trees created in error checking and code generation of the for-loop example in Figure 1; it also illustrates a potential problem with this approach. For two nested for-loops, the original tree with body *s* is shown on the left (the loop variables and bounds are not shown). The propagation of a typing environment for error checking (`errs`) is indicated by `env` that is propagated along those edges. The squiggly edges facing right indicate the translation of the loops, with the clouds representing the while-loop code containing *s*. Note that the translation of the outer for-loop includes the inner for-loop, an approach used in the *forwarding* technique [26] used in ableC to handle independent language extensions. This avoids inappropriately translating-away constructs from other independent extensions. The assembly language translation process to construct `asm` (which may also depend on types and thus `env`) is dispatched from the for-loops to their translations and this process takes place on right-most trees, the ones whose nodes have an `env` propagated down (or over) to them. This will provide two copies of *s* with an environment, but the middle two instances of *s* would not be constructed nor visited in these two compiler tasks.

In some cases, this incremental translation may even result in an exponential number of trees to traverse. This can happen in some instances of type-based operator overloading, in which the type of the result of an overloaded construct is not determined explicitly, but is instead computed on the overloaded construct's translation, through forwarding. Consider an overloaded negation operator ~ and the type-checking of the expression `~ (~ (~ e))`. Following this pattern, type checking results in creating and traversing an exponential number of trees, 8 in this case, as illustrated in Figure 3. The overloaded ~ operator, represented as `neg`, will type-check its sub-expression since that type is used to determine the translation, the `neg'` nodes in Figure 3. The type of a `neg` expression is determined by its `neg'` translation and thus we need to type check *e* under both operators. An enclosing (middle) `neg` expression queries its child to determine its translation and that translation will do the same. The results in type checking *e* 4 times. Another enclosing negation operator repeats the process and then *e* is checked 8 times, an exponential growth in the number of trees created and traversed results. In fact, this phenomenon can also occur with seemingly predictable constructs like the for-loop when, *e.g.* another extension uses both `errs` and `asm` results from a child for-loop to compute `asm`.

A second problem faced in specifying language features in this manner is the need to explicitly manage the flow of contextual information down to the components of the new language feature. In some cases this can be difficult, in others only tedious. In Figure 2, a `forLoop` construct extends its incoming environment *env* to include the declaration of the loop variable. This extended environment is passed as env_1 to the nested loop, which does the same to pass env_2 for the body `s`. It is important to make sure that the environments supplied by a production are the same as, or at least compatible, with those determined on the translation. Here, this entails ensuring that env_1 (or env_2) is compatible with env'_1 (or env'_2) as determined on the translation. As we will see, this is straightforward for a `forLoop`, but in the more sophisticated language extensions found in ableC this can be more difficult.

A solution to both of these problems is for component trees to be shared by the non-core feature and its translation. This is depicted on the left in Figure 4, in which the loop

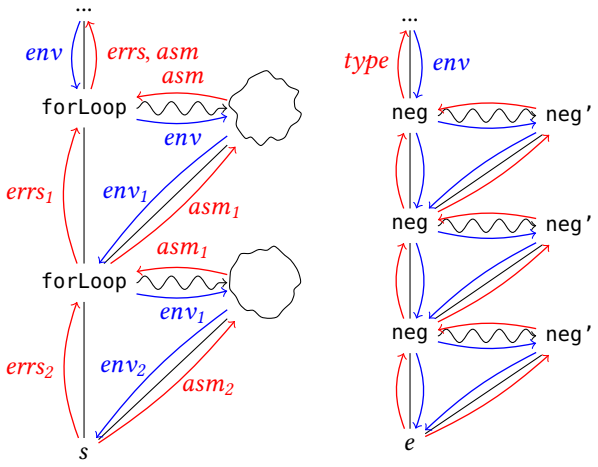

Figure 4. Alternative versions of the trees in Figure 2 and Figure 3 in which decorated children are shared between the forwarding and the forwarded-to trees.

body *s* exists only once and the `env` context is specified by the translation. (The lower and upper bounds of the for-loop are shared in a similar way.) Thus, the for-loop construct can still access type information on those expressions and generate appropriate error messages when those expressions do not have an integer type. Tree sharing also eliminates the exponential number of trees in the negation example, as seen on the right in Figure 4. The ableC system frequently uses a crude approach to sharing that avoids this duplication of trees but limits the extensibility of the language feature, the primary goal of ableC.

In addition to these concerns, there are also instances when a new language construct needs to supply contextual information to its components that differs from that supplied by the translation. For example, a pretty-printing task may provide an indentation level to its sub-terms and this value would differ for the loop body under these two constructs. If for a language feature the translation is not "macro-like" (as in the for-loop example) but instead must be computed from semantic information, such as the types of sub-terms, then these explicit contextual specifications are required. Additionally, there are situation in which sharing of sub-components may not be feasible and two versions of the tree are needed. For example in translating "`repeat` *body* until *cond*" to "*body* `; while (not` *cond*`) do` *body*" the *body* tree cannot be shared in both places; some attribute (*e.g.* a data-flow analysis) may need to have different values for each instance of *body*.

This paper examines this problem and poses solutions in the context of attribute grammars (AGs) with *forwarding* [26] — a technique that constructs translations and automatically copies contextual information to the translation. Computed semantic information on the translation is automatically copied back to the original construct, when it is not overridden (*e.g.* error messages) by an explicit definition on the "forwarding" construct.

The primary contribution of the paper is a new mechanism for sharing trees and their attribution under a new (forwarding) construct and its translation (forwarded-to) construct. In attribute grammars, a production defines synthesized attributes for the left-hand side nonterminal and inherited attributes for the right-hand side nonterminals. Forwarding provides default/implicit equations for synthesized attributes. We extend this so that for shared trees, forwarding can now do the "other half" of this work and now provide default-/implicit equations for inherited attributes for right-hand nonterminals too. Critically, this mechanism does not limit the extensibility of the language like the crude mechanism currently used in ableC. This is discussed in Section 3 after Section 2 continues the discussion of the shortcomings of the existing approach to forwarding. Section 3 also introduces *translation attributes*, a means for sharing context when translation trees are constructed over a number of productions in a higher-order attribute.

Section 4 validates the techniques by implementing them in the Silver [25] attribute grammar system and applying them in the large ableC specification and several extensions to it, finding that around 80% of the uses of the crude non-extensible technique (and all exponential cases) could be replaced by the new extensible approach.[1]

Section 5 describes how the modular well-definedness analysis [12] can be extended to handle this new feature to ensure that there will be no missing or duplicate equations in an attribute grammar composed from independently-developed language extensions. It also discusses challenges in ensuring non-circularity with this approach.

We discuss limitations of this approach, and alternatives to it, in Section 6 before discussing related in Section 7 and future work and concluding in Section 8.

2 Background

In this section we provide background on attribute grammars, forwarding [26], its use in extensible languages, its limitations, and its realization in the Silver AG system [25]. Background on the modular well-definedness analysis [9, 12] and its extension in this paper is discussed in Section 5.

2.1 Attribute grammars and Silver

Attribute grammars are a declarative formalism for specifying the semantics of context-free languages [15, 16] and can be formally defined as a four-tuple $\langle G = \langle NT, T, P\rangle, A, O, E\rangle$ where G is a context free grammar with nonterminal symbols NT, terminal symbols T, and production rules P. A is a set of synthesized (A_S) and inherited (A_I, $A = A_S \cup A_I$)

[1] Silver, ableC, extensions and other examples are available at https://melt.cs.umn.edu and archived at https://doi.org/10.13020/badh-qf44.

attributes that may decorate nodes of trees in the language of G, and O is a mapping of which attributes in A *occur on* which nonterminal symbols in NT. E is the set of equations on productions, that define the values of attributes on trees.

Since AGs process abstract syntax, as opposed to specifying concrete syntax, a grammar G supports a wider variety of types for tree nodes than simply nonterminal and terminal symbols. Thus, productions in P have a signature of the form $x_0 :: NT_0 ::= x_1 :: V_1 ... x_n :: V_n$, with $n \geq 0$ in which V includes NT, T, primitive types such as integers, and others. These signature items are labeled with names so that nodes in a syntax tree can be referred to by these labels instead of their position in the production.

Attribute grammars have been extended in a wide variety of ways since their introduction to better support the specification of tasks common in language implementation. For example, higher-order attributes [29] hold tree values that are passed to new locations in the syntax tree where they are then provided with inherited attribute (that is, decorated) so that synthesized attributes can be computed on them. Another commonly used extension is reference [7] or remote [1] attributes. These can be seen as pointers, or references, to the root node of remote decorated trees somewhere in the syntax tree from which attributes can be accessed.

In Silver decorated trees in reference attributes are known to have been provided with a set of inherited attributes called a *reference set*. When decorated tree types are written as `Decorated` *NT* this set is, by default, the inherited attributes occurring on *NT* that were declared in the same grammar module as *NT*. The reference set can be given explicitly to override this default; *e.g.* `Decorated Expr with {env}` identifies the environment attribute `env` as sole attribute in the reference set for this type of decorated expression.

Figure 5 show an implementation of the for-loop construct as seen in Figure 1 in the Silver AG system. Declarations of attributes, and their occurrences, are not shown but can be inferred from the `forLoop` production. This production computes a synthesized `errors` attribute on the left-hand side `s`, defines the inherited environment attribute `env` on the three child trees, computes a local fresh variable `upperVar`. This is used in the construction of the while-loop translation, seen in Figure 1, that the for-loop will forward to. The productions `decl` for declaring and initializing variables, `seq` for statement sequencing, etc. should be clear from the example in Figure 1. The `forLoop` production takes (undecorated) terms of type `Expr` and `Stmt` in constructing the syntax tree. In the body of the production (in the curly-braces) these labeled terms are decorated by inherited attribute equations, *e.g.* lines 7–9, and thus the labels refer to trees of type `Decorated Expr` and `Decorated Stmt`. Similarly, local tree-valued *production attributes* can also be declared and defined as an undecorated nonterminal type (*e.g.* `Expr`) and decorated using inherited attribute equations in a production body to be typed as decorated (*e.g.* `Decorated Expr`).

```
1  production forLoop  s::Stmt ::=
2   iVar::String lower::Expr upper::Expr body::Stmt
3  { s.errors =
4      checkInt(lower.type, "loop lower bound") ++
5      checkInt(upper.type, "loop upper bound") ++
6      lower.errors ++ upper.errors ++ body.errors;
7    lower.env = s.env;
8    upper.env = s.env;
9    body.env  = addEnv(iVar, intType(), s.env);
10   local upperVar::String = freshName(s.env);
11   forwards to block(seq(
12     decl(iVar, intType(), new(lower)),
13     seq(decl(upperVar, intType(), new(upper)),
14       while(intLt(var(iVar), var(upperVar)),
15         seq(new(body), assign(iVar,
16           intAdd(var(iVar), intConst(1))))))));
17 }
```

Figure 5. A `forLoop` implementation. The children are explicitly decorated with the environment to support error checking, and are decorated again in the forwarded to tree.

2.2 Forwarding and extensible languages

Forwarding is a technique developed to support the modular definition of languages [26] and has been used extensively in the ableC extensible C compiler and a wide variety of composable extensions to it [10, 11]. Any queries for synthesized attributes on the production's left-hand side nonterminal that are not explicitly defined by equations are "forwarded" to the `forwards to` tree to be answered there. Likewise, any queries for inherited attributes on the forwarded-to tree are passed back to production's left hand side and their values are retrieved from there. In Figure 5, `s` defines a value for the `errors` attribute using the function `checkInt` that reports a message when a type is not an integer. This takes precedence over the value for `errors` on the while-loop construct, thus providing proper error messages that reference the code written by the programmer, not the code to which it translates. A query to `s` for an assembly language translation `asm` attribute would automatically and implicitly copy the value from of `asm` from the forwards-to tree back to `s`. Any queries of an inherited `env` attribute on the outermost `block` construct in the forwards-to tree would get is value from the `env` attribute passed down to the left-hand side symbol `s`.

Forwarding supports the automatic composition of independent extension specifications. If another extension defines, *e.g.* a new translation to Web Assembly in a `wasm` attribute, then all computations involved in that effort take place on the forwards to tree and automatically provide a value for `wasm` for `s`, even though the author of the `forLoop` extension knew nothing of this Web Assembly extension.

```
1  production neg   e::Expr ::= n::Expr
2  { n.env = e.env;
3    forwards to case n.type of
4      | intType()  -> intNeg  ( new(n) )
5      | boolType() -> boolNeg ( new(n) )
6      | _ -> errorExpr ("incorrect types")
7      end;  }
```

```
1  production decExpr
2  e::Expr ::= de::Decorated Expr with {env}
3  { e.type = de.type;
4    e.errors = de.errors;  }
5
6  production neg  e::Expr ::= n::Expr
7  { n.env = e.env;
8    forwards to case n.type of
9      | intType()  -> intNeg  ( decExpr(n) )
10     | boolType() -> boolNeg ( decExpr(n) )
11     | _ -> errorExpr ("incorrect types")
12     end;  }
```

Figure 6. The exponential `neg` production (top) and the efficient but crude "decExpr" hack (bottom).

2.3 Limitations of forwarding

While forwarding provides implicit attribute definitions for synthesized attributes for a production's left-hand side symbol, it provides no support for the "other half" of what equations associated with a production do: provide values of inherited attributes to the child trees. This is shown in Figure 5 where equations for `env` are required for child trees since their synthesized attribute `errors` (whose computation depends on an environment) is demanded on line 6. An important consideration when overriding `errors` on `forLoop` is that any problems in the forward tree should still be reflected in the new `errors` equation. To ensure this, the environment given to `lower`, `upper` and `body` must match the environment computed through the equations of the productions enclosing these children in the forward tree; *e.g.* `body` must receive an `env` containing `iVar` bound to the appropriate type. This requires extension developers to familiarize themselves with host language details in order to write the appropriate equations. Doing so may be especially burdensome in a more sophisticated host language where the inherited dependencies of `errors` could be more than just `env`.

Performance is another concern; see the duplication of for-loop (Figure 2) and negation trees (Figure 3). In the specification of the forwards-to tree for a for-loop in Figure 5 the child trees `lower`, `upper`, and `body` must be replicated (using `new`) This "undecorates" these trees, retrieving the original terms prior to decoration, and re-decorates them with new attribute values under the forwards-to tree. For the for-loop in Figure 2 this is not so expensive, but it is a significant problem with the negation operator in Figure 3. The specification of the overloaded negation operator is given at the top of Figure 6. It queries its child term's type (`n.type`) to determine which type-specific negation production to forward to. Since there is no explicit equation for `e.type` that value is determined on (and copied from) the forward tree, resulting in the duplication in Figure 3.

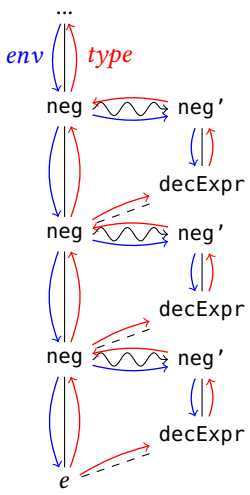

Figure 7. The efficient but non-extensible implementation of overloaded negation from the bottom of Figure 6.

The specifications at the bottom of Figure 6 demonstrate the crude "decExpr" hack that avoids the exponential duplication of trees. The `decExpr` production wraps up a `Decorated` expression tree that has been already provided with its `env` attribute (written `with {env}`), thus allowing the evaluation of the `type` and `errors` attribute on `de`. This production is used to wrap up the decorated child `n` in the optimized `neg` production instead of using `new`. This results in the tree shown in Figure 7 in which each negation child is shared between the original and forwarded-to trees.

While this is efficient, it severely limits the extensibility of the language. Recall the extension adding equations for a synthesized `wasm` attribute to host language productions. It would define `wasm` on `intNeg` and `boolNeg` but any new inherited attributes needed for this computation will not be propagated down past the `decExpr` node. The needed inherited attribute equations cannot be added to the host-language `decExpr` production since they would have no effect — its child is a reference to a tree that was decorated elsewhere, and inherited attribute equations are not permitted here. This technique is used frequently in the ableC specification and, while efficient, limits the kinds of language features that can be developed as composable language extensions.

3 Forwarding with Tree Sharing

Here we describe a new "tree-sharing" operator `@` that allows trees, and the specification of inherited attributes, to be shared between a forwarding and a forwarded-to tree, thus avoiding the duplication and redecoration seen in Figure 2 and Figure 3 and instead producing shared trees like those in Figure 4. We also describe different scenarios in which this can be used.

```
1  production forLoop  s::Stmt ::=
2   iVar::String lower::Expr upper::Expr body::Stmt
3  { s.errors =
4     checkInt(lower.type, "lower bound") ++
5     checkInt(upper.type, "upper bound") ++
6     lower.errors ++ upper.errors ++ body.errors;
7    local upperVar::String = freshName(s.env);
8    forwards to block(seq(
9      decl(iVar, intType(), @lower),
10     seq(decl(upperVar, intType(), @upper),
11       while(intLt(var(iVar), var(upperVar)),
12         seq(@body, assign(iVar,
13           intAdd(var(iVar), intConst(1)))))))); }
```

Figure 8. An alternative version of Figure 5 in which children are shared with the forward tree, avoiding the need to specify inherited `env` equations.

3.1 Sharing with a static forward tree

In order to achieve the pattern of sharing seen in Figure 4, we introduce the *tree-sharing* operator `@`, which takes a decorated tree and wraps it as an undecorated term. Decorating this term simply yields the original tree, updated with any newly-supplied attributes added; thus the syntax `@a` can be read as "`a` gets decorated with more attributes here." This operator can be seen as a improved, built-in version of `decExpr` and similar "wrapper productions", except that there is no intermediate node for `@` in the decorated tree, like the `decExpr` one seen in Figure 7. The tree-sharing operator can be used in specifying the forward for the `forLoop` production, as seen in Figure 8. With the nested `forLoop` scenario, this gives rise to the tree in Figure 4 instead of Figure 2; now the innermost statement *s* is only decorated once.

When a shared child or local appears beneath statically specified productions in the forward tree, inherited attributes supplied to the child/local by these productions can be utilized in the original forwarding production. This effectively permits child inherited attributes to be implicitly computed through forwarding. For example on lines 4–6 of Figure 8, `type` and `errors` can be accessed on the children without supplying explicit equations for `env` since, as seen on the left in Figure 4, `env` is defined for those trees by the forward tree.

Silver uses a demand-driven approach to attribute evaluation [8], which now presents some complications. Traditionally, one knows what equations will be used to compute inherited attributes on a tree before any attributes are evaluated. This is now no longer the case, as inherited attributes supplied to a child shared in a forward tree are only defined when the portion of the forward tree containing the child is demanded. For example in Figure 8, the equation for

```
1  production neg   e::Expr ::= n::Expr
2  { n.env = e.env;
3    forwards to case n.type of
4      | intType()  -> intNeg  ( @n )
5      | boolType() -> boolNeg ( @n )
6      | _ -> errorExpr ("incorrect types") end;  }
```

Figure 9. An implementation of operator overloading, in which the forward tree is determined based on the operand types. Computing these types requires supplying the environment to the operands.

`s.errors` does not directly depend on forwarding; demanding `lower.errors` seemingly would not cause the forward tree to be created and decorated, which would lead to a missing equation for `env` on `lower`.

To avoid this, any use of the original child tree must demand the corresponding portion of the forward tree. This is conceptually like pattern matching on the forward tree, with a pattern that mirrors the term containing the child. For example, the access of `lower.errors` could be translated as

```
1  case forward of
2  | block(seq(decl(_, _, lower))) -> lower.errors
3  end
```

In reality, this can be implemented more efficiently than pattern matching, as the forward tree is known to have been built with these constructors and we do not need to check that it has the expected shape. Note that this problem would not exist if a similar approach of decoration through a shared tree was used in an ordered attribute grammar [14]; an attribute supplied through sharing (like `env`) could be fully computed before being used on the shared tree by another attribute (like `errors`.)

In prior versions of Silver, children and locals could be referenced with an undecorated type, implicitly un-decorating trees in these cases. For example, the calls to `new` on lines 4–6 of Figure 5 could have been omitted. We now recognize this to be a language design flaw, as children can be inadvertently un-decorated, and included in the forward tree to be decorated again, without any indication of a potential problem. To address this, we simplify the type semantics of Silver so that any reference to a child or local tree gives a decorated type. This requires one to explicitly write `new` or `@` when incorporating a sub-tree that has previously been decorated into a new term.

3.2 Dynamic forwarding

Sometimes, a child may appear in the forward tree under different productions, depending on the result of some analysis. This is the case for the negation operator from Figure 6, where `n.type` is computed to determine the target

```
1  production neg   e::Expr ::= n::Expr
2  { local nVar::String = freshName(e.env);
3    local impl::Expr = case n.type of
4      | intType()  -> intNeg  ( var(nVar) )
5      | boolType() -> boolNeg ( var(nVar) )
6      | _ -> errorExpr ("incorrect types")   end;
7    forwards to let_(nVar, @n, @impl);  }
```

Figure 10. An alternate version of Figure 9, in which the portion of the forward tree containing the child `n` is static.

production. Computing `type` on an expression depends on `env`, which must be supplied to `n`; to avoid a circularity, this must be done with an explicit equation rather than through forwarding.

However, one would still wish to share `n` in the forward tree, to avoid the exponential explosion seen in Figure 3; this can be done using the tree sharing operator as seen in Figure 9. For this to be possible, the writer of a forwarding production such as `neg` must ensure that the values of any explicit inherited equations on a shared child match the values that would otherwise be supplied through forwarding.

3.3 Partially dynamic forwarding

Often, an intermediate approach between static and dynamic forwarding is possible: children can be shared (and receive attributes) beneath a static portion of the forward tree, while other portions of the forward tree are computed dynamically. For example in Figure 10, the operand to `neg` can be bound to a fresh temporary variable in a let-expression. The implementation, dynamically determined from the type of `n`, can then refer to the variable instead of using `n` directly. Lazy evaluation means that the portion of the forward tree containing `n` can be decorated with `env` to compute `type`, before `impl` is computed. Note that `impl` is also marked as being shared, since it is a local that gets decorated implicitly.

Note that without care this approach can give rise to circularities, between computing an analysis on a child that is needed to determine part of the forward, and decorating the forward tree to determine inherited attributes on the child. To avoid this one must sometimes supply some inherited equations explicitly, which take precedence over equations supplied in the forward tree.

For example, the ableC-closure extension [10] introduces lambda functions, *e.g.* `lambda (int x) -> x + y`, where free variables such as `y` referenced in the body can be captured. To implement this, a lambda function is implemented as a function pointer, paired with a struct containing the values of captured variables. Thus, the above lambda expression would forward to a function pointer to the following function that is lifted to the global scope along with the following `struct` declaration:

```
1  var res : bool = table { b1 && b3 : T F
2                           ~ b2     : T *
3                           b2 || b3 : F T };
```

```
1  var res : bool =
2    let _v0 : bool = b1 && b3 in
3    let _v1 : bool = ~b2 in
4    let _v2 : bool = b2 || b3 in
5      (_v0 && _v1 && ~_v2) || (~_v0 && _v2);
```

Figure 11. A simple language extension for condition tables (top), provides an alternate concise notation for complex boolean expressions (bottom)

```
1  struct _lam_env_19 { int y; };
2  int _lam_fn_19(struct _lam_env_19 _env, int x) {
3    const int y = _env.y;
4    return x + y;  }
```

The free variables from the body to be captured are computed as a synthesized attribute `freeVariables`, which on expressions depends on `env`. However, the `env` given to the body in the forward of the lambda production depends on the variable definitions (*e.g.* line 3 in the above) generated from the free variables. This circularity can be avoided by the lambda production supplying an explicit `env` equation to its body expression. For correctness, this equation must match the `env` supplied through forwarding, in this case by making all captured variables constant.

3.4 Computing a forward over multiple productions

Sometimes, an extension may introduce its own nonterminals to provide richer syntax, and the computation of the translation it will forward to is spread across the productions for these new nonterminal symbols. For example, the condition tables extension, seen at the top of Figure 11, provides convenient syntax for writing complex Boolean expressions. A condition table expression is true if there is a column where the expression is true for every row with a `T`, and is false for every row with an `F`, while `*` indicates that we don't care if that expression is true or false. This expression translates to the code seen in the bottom of Figure 11, creating a let-binding for every expression, with the conditions translated into conjunctive normal form as the body.

A portion of the implementation of this extension is seen in Figure 12. Table rows are represented by the `TRows` nonterminal (line 7), with an inherited attribute `conds` to construct the needed Boolean result expression. We require in the syntax of the extension that the table has at least one row, such that `rs.conds` is non-empty in `nilRow`. On line 16, the `trans` attribute then wraps the result in the needed let bindings for the row expressions.

```
1  production condTable  e::Expr ::= rows::TRows
2  { e.errors = rows.errors;
3    rows.conds = [];
4    forwards to @rows.trans;  }
5  inherited attribute conds::[Expr];
6  translation attribute trans::Expr;
7  nonterminal TRows with errors, conds, trans;
8  production consRow  rs::TRows ::=
9    e::Expr tf::TruthFlags rest::TRows
10 { rs.errors = e.errors ++ tf.errors ++ rest.errors
11     ++ checkBoolean(e.type, "row expression");
12   local eVar::String = freshName(rs.trans.env);
13   tf.rowExpr = var(eVar);
14   rest.conds = if null(rs.conds) then tf.rowConds
15     else zipWith(andOp, rs.conds, tf.rowConds);
16   rs.trans = let_(eVar, @e, @rest.trans);
17 }
18 production nilRow  rs::TRows ::=
19 { rs.errors = [];
20   rs.trans = foldr1(orOp, rs.conds);  }
```

Figure 12. A portion of the implementation of condition tables, computing a forward tree containing children decorated across multiple productions using a translation attribute.

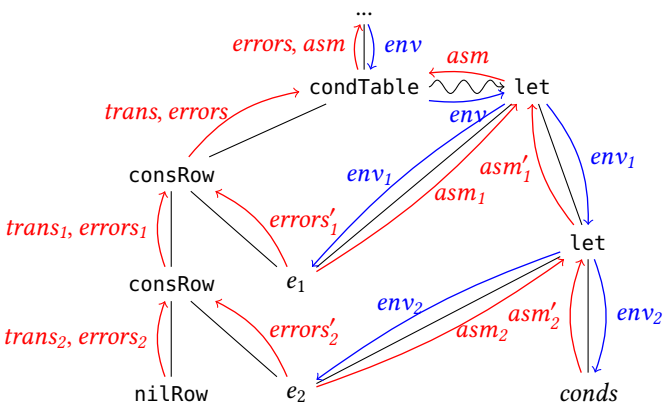

Figure 13. A tree corresponding to the condition tables extension in Figure 12. `trans` and `errors` is computed on the `TRows` nodes on the left, while `env` and `asm` are computed on the decorated form of `trans` on the right.

As seen previously with `neg` in Figure 10, we would like to decorate these host-language `Exprs` with inherited attributes supplied through their translation. However, `consRow` is not a forwarding production since its left-hand side is not a host language nonterminal. If `trans` were an ordinary higher-order attribute, we would construct an undecorated translation term and decorate it in the forward of `condTable`. This would then decorate the condition expressions in the table rows again, thus returning to the situation in Figure 2.

Instead, we would like to create the pattern seen in Figure 13. Here we first construct the translation of the table rows into `let`-expressions, and decorate it as the forward of `condTable`. The environment flows down the forward tree to the condition expressions, where it is used to compute `errors`, which are collected up the original `TRows` tree.

This can be achieved using *translation attributes*, which are synthesized attributes that serve as decoration sites for terms, similar to locals. Translation attributes are similar to higher-order attributes in that their equations create undecorated terms; the declared type of a translation attribute must be a nonterminal. However like reference or remote attributes, they hold decorated trees and may be supplied with inherited equations. On line 4 of Figure 12, accessing `rows.trans` gives back a `Decorated Expr`. On line 10, `consRow` can access `e.errors`, which depends on `e.env`, using the `env` supplied through the equation for `rs.trans`.

Synthesized and inherited attributes occurring on `Expr` may be treated like additional synthesized and inherited attributes occurring on `TRows`; for example, the `consRow` production uses `rs.trans.env` to compute `eVar`, which must have been supplied to `rs.trans` by the parent of this production. We can again use the `@` tree-sharing operator to indicate that the tree constructed by a translation attribute should be shared with (and receive inherited attributes from) another decorated tree. In `condTable`, the `env` supplied to `rows.trans` (through the use of `@` and forwarding here; once could also write `rows.trans.env = e.env;`) flows down through the `Expr` tree built by `rows.trans`. Thus in `consRow` one can access `e.errors`, which depends on `e.env`, using the `env` supplied through the equation for `rs.trans` on line 16.

Demand-driven evaluation creates complications for translation attributes, like was seen with sharing children in Section 3.1. Any use of a tree shared in a translation attribute equation must demand the decoration of the tree constructed by the attribute, which may involve recursively demanding the decoration of the translation attribute from further up the extension tree. For example, accessing `errors` from e_2 in Figure 13 must ultimately demand the forward tree from its root decoration site in the forward of `condTable`. To achieve this, the implementation involves another implicit inherited reference attribute to pass corresponding portions of the translation tree down from its root decoration site.

4 Evaluation

We evaluated the utility of this new approach to forwarding in the ableC [11] host language specification of C. There are many different extensions to ableC [10, 11, 17] and we evaluated 10 non-trivial extensions that are representative of the other ableC language extensions. The full list of extensions and detailed results of the evaluation can be found

in Appendix A. Overall, 94% of uses of decorated wrapper productions like `decExpr` in Figure 7 could be replaced by the tree sharing operator, and those that could not were not needed to avoid exponential decoration. These changes allow the removal of many complex and now-unneeded inherited attribute equations, making the extension specifications significantly easier to understand and maintain.

All unary and binary operators in ableC, as well as some expressions like function call and array index, support operator overloading. Each operator has a production that forward based on the types of the children; these explicitly supply inherited attributes and wrap the decorated children in `decExpr`, similar to `neg` in Figure 9. In total, `decExpr` is used 168 times in the ableC specification; all of these uses could be replaced with uses of the tree sharing operator.

Across the 10 considered ableC extensions, there are 74 instance of decorated wrapper productions. Of these, 39 appear in a statically-determined forward tree, where the tree-sharing operator can be used to avoid some inherited equations, and 5 appeared in a dynamically-forwarding production similar to overloading, where inherited equations are still needed. In 30 cases, wrapper productions were used in a translation passed as an attribute to be decorated elsewhere. Of these, 15 could be easily replaced using translation attributes.

The ableC-Prolog extension uses a combination of multiple higher-order synthesized and inherited attributes for constructing a translation, where some host language children may appear in the trees constructed by more than one attribute. This pattern is not amenable to translation attributes; thus there are are 15 uses of decorated wrapper productions in the Prolog extension which could not easily be replaced. However, the constructs introduced in this extension are not typically nested, and thus would not suffer from exponential recomputation if these subtrees were decorated twice.

In some ableC extensions, *e.g.* condition tables, we found instances of tree re-decoration that had not been addressed with wrapper productions. We have not yet attempted to identify all instances of trees being re-decorated that could potentially be addressed with tree sharing. This is because these specifications were developed with a prior version of Silver, with the problematic implicit undecoration semantics that we mentioned in Section 3.1. Changing the type rules to require explicit undecoration or sharing creates an error flagging every potential place where a tree could be re-decorated; however this constitutes a major breaking change to all existing Silver code, and we had not yet completed this change as of performing this evaluation.

For this reason we have not attempted a comprehensive refactoring of ableC and its extensions to use the new sharing mechanisms, or quantified the number of unaddressed re-decoration issues. Instead, we refactored 3 ableC extensions to use the new sharing mechanisms: closures, condition-tables, and algebraic datatypes. In all cases this led to simpler code and the removal of inherited attributes and equations. In the datatypes extension, the use of translation attributes to translate pattern matching saved 36 lines of specification (out of a total of 900.)

The tree-sharing operator provides a small optimization over the old approach of wrapper productions, as the new approach does not require introducing an extra node in the decorated tree, as seen in Figure 7 vs Figure 4. The performance was evaluated by comparing the runtime of building the examples for the closure and datatype extensions before and after refactoring; the refactored versions yielded a roughly 4% speedup. We did not attempt to compare the performance of specifications with exponential re-decoration behavior to ones in which sharing is used, as prior to refactoring these were clearly unusable for nontrivial programs.

Overall, tree sharing has proven to be a very beneficial technique, with significant improvements in code quality, and avoiding excessive recomputation of analyses without limiting extensibility.

5 Modular Well-Definedness

Kaminski and Van Wyk [12] proposed a modular well-definedness analysis for attribute grammars with forwarding, to ensure that all potentially needed attribute equations are present in compositions of independently-designed language extensions. This analysis is based around the idea of constructing flow graphs for productions, as in Knuth's original analyses [15]. *Flow types* are inferred for every occurrence of a synthesized attribute on a nonterminal, consisting of the set of inherited attribute dependencies. Attribute equations are checked against the production flow graphs and flow types, ensuring that all needed equations are supplied.

The data structures and algorithms used in Kaminski and Van Wyk [12]'s analysis, used in Silver, must be slightly extended to accommodate the new features proposed here. While space limits preclude a full presentation of the extended analyses here, we present the issues at hand and give some intuition about the required changes below.

5.1 Avoiding duplicate equations

Since we are adding the ability to supply inherited equations to the same tree from multiple sites, we must avoid there being multiple equations for the same attribute when some precedence between them cannot be determined. First, the operand to the `@` tree-sharing operator must correspond to a *decoration site* where inherited attribute equations could ordinarily be specified, *i.e.* a nonterminal child, local, or instance of a translation attribute on a child or local.

If an explicit inherited equation is given to a child in a forwarding production, in addition to an equation through a production in the forward tree, then the explicit equation

takes precedence. This is analogous to the explicit synthesized equations on the forwarding production taking precedence over those in the tree forwarded-to. We must also ensure that if a child is shared, it only appears in one place, to avoid multiple competing equations. This means that a shared child can only appear in the original production, and not in independently-introduced aspect productions. Similarly, a local tree can only be shared once and in the same module as the local, and a translation attribute instance can only be shared once per production in the same module as the attribute occurrence. It is permitted for the same tree to be shared in multiple mutually exclusive positions, such as separate branches of a pattern match as seen in Figure 9.

The expression in which a shared child appears also must not decorate the term containing the child more than once, for example a shared tree cannot appear in an arbitrary higher-order attribute equation, which may be arbitrarily used and re-decorated elsewhere. The tree-sharing operator must only appear in unique contexts, which can be a forward equation, local equation, translation attribute equation, or as an operand to a production call or conditional expression in a unique context. This is enforced by a straightforward syntactic analysis, termed the *uniqueness analysis*.

A more subtle issue exists if a production in the forward equation were to un-decorate and re-decorate its own child, perhaps intentionally to perform an analysis in a different environment. The behavior of calling `new` on a tree in Silver is to simply return the original term that was decorated to create the tree. If the term provided to this production contains a wrapped child from use of the tree-sharing operator, this may result in the child being decorated twice.

Instead, we change the semantics of un-decoration in Silver to perform a deep-copy of the term that was decorated, such that calling `new` on a decorated tree never returns a term containing a wrapped tree. Thus in the above scenario, the un-decorated and re-decorated tree does not share any subtrees. As an optimization, all constructed terms track whether they contain a wrapped tree, such that the deep copy operation only needs to happen down to the level of any wrapped subtrees.

5.2 Enforcing effective inherited completeness

Recall the intuition given in Section 3.2 for the runtime semantics of tree sharing in terms of pattern matching. The treatment of the tree-sharing operator in the modular well-definedness analysis is also similar to pattern matching, as discussed fully in previous work [9]. A new sort of flow vertex is introduced corresponding to unconditionally-decorated sub-terms of a forward or local equation. The existing analysis has a notion of *flow projection stitch points*, used to update a production's flow graph with edges corresponding to inherited equations from productions referenced in patterns. Flow projection stitch points are also used here, to add edges for inherited equations between sub-term vertices.

A synthesized attribute may now depend on an inherited attribute being supplied to a translation attribute on a tree. Thus, flow types are extended to include inherited attributes occurring on translation attributes on a nonterminal, in addition to inherited attributes occurring on a nonterminal.

For inherited attribute equations to be reliably supplied to a child through forwarding, the requirements for the term in which the child appears are somewhat stricter than imposed by the uniqueness analysis. The child must be decorated *unconditionally*, meaning that it cannot appear under any conditional expressions. For example as seen in Figure 9, `n` can be shared in the forward tree, but we cannot rely on any inherited equations supplied through forwarding.

The checks performed on equations using the inferred flow types and production flow graphs are essentially unchanged, except that we do not check for the presence of inherited equations within a production for a shared, unconditionally decorated child or local. There is an additional check required for inherited override equations on children or locals that are shared, even conditionally: the dependencies of the override equation must not exceed the dependencies of the remote equation. This is because the production constructed with the shared child will be checked with the flow graph created from the inherited equations that it supplies; if we override this equation with one that has additional dependencies, this may lead to additional transitive dependencies that were not checked, and potentially missing equations.

5.3 Limited feasibility of circularity analyses

Kaminski and Van Wyk [12]'s modular well-definedness analysis does not include a non-circularity check. This is because in practice, strict non-circularity between attributes is overly conservative; laziness means that cycles often are not present in the actual evaluation even when a strict analysis could not prove their absence. Furthermore circularity between portions of trees is often useful, such as in building the environment for mutually recursive bindings. For these reasons, attribute grammar systems such as Silver, JastAdd [4] and Kiama [21] dispense with a non-circularity analysis.

Circularities between different portions of trees have proven especially useful with tree sharing, as seen in Figure 10. We have found that this pattern does frequently give rise to actual cycles, between an inherited attribute on a shared child and a dynamic portion of the forward tree that may affect the child's attribute. Since we don't have an analysis capable of detecting these problems, these issues are typically found through crashes when developing an extension; however we have found them to be easy to diagnose and resolve by adding explicit inherited equations. These problems also typically do not appear from composing independent extensions, because the problematic dependencies involve productions explicitly specified in the forward tree. While a more sophisticated analysis could be useful for finding these issues, it is unclear if such an analysis is feasible.

6 Discussion

This section provides a discussion of tree sharing and the computation of inherited attributes through forwarding, and how this work relates to other aspects of Silver.

6.1 Specializing inherited attribute equations

A nice feature of forwarding is the freedom to specialize synthesized attributes to the new language construct by writing explicit equations for them. This allows one to report error messages specific to the new feature or define the type attribute on a new expression, *e.g.* of a extension introducing list literals may define its type to be a list type. In Section 3.3 we saw an example of needing to write explicit inherited equations to break circularities with the closure extension. These equations always had the same value as what would later be computable in the forward tree. Are cases when we want to specialize an inherited attribute with a different value, as seen with errors? We expected the new ability to specialize inherited attributes to be similarly helpful, but their actual benefit was something else. Contextual information, *e.g.* an environment mapping names to types, is some data structure containing terms for host language nonterminals, such as a `Type` nonterminal defining structured types. Thus new extension types, *e.g.* list types, naturally arise in the name bindings. It turns out there is not much need to specialize an environment, or similar inherited attributes, on the forwarding production. More frequently, we find one writes these explicit equations to break cycles in extensions with sophisticated patterns of forwarding. Since these equations are not so often intended to specialize inherited attribute values, it is also easier to ensure that the values given are compatible with those provided on the forwarded-to tree.

6.2 Why autocopy is a misfeature

Past versions of Silver and earlier presentations of forwarding [25, 26] featured *autocopy attributes*, a form of inherited attributes that were implicitly copied down the tree to children. This is convenient for attributes such as an environment that generally flow down the tree. However autocopy attributes are incompatible with the new approach to tree sharing, as we often want an environment to be supplied through forwarding, and autocopy would supply an undesired implicit copy equation for the child.

In fact, we had already recognized autocopy as a source of bugs due to undesired equations: in developing attribute grammar specifications, we sometimes use a "flow-type-driven development" approach, adding needed equations as they are flagged by the well-definedness analysis. Autocopy attributes suppress these errors by introducing implicit (and often incorrect) equations. For these reasons we have removed support for autocopy, and replaced it with a mechanism to specify where copy equations should be generated for an inherited attribute.

```
1  production forLoop  s::Stmt ::=
2   iVar::String lower::Expr upper::Expr body::Stmt
3  { local localErrors::[Message] =
4      checkInt(lower.type, "lower bound") ++
5      checkInt(upper.type, "upper bound") ++
6      lower.errors ++ upper.errors ++ body.errors;
7    local upperVar::String = freshName(s.env);
8    forward fwrd = block(seq(
9      decl(iVar, intType(), @lower),
10     seq(decl(upperVar, intType(), @upper),
11       while(intLt(var(iVar), var(upperVar)),
12         seq(@body, assign(iVar,
13           intAdd(var(iVar), intConst(1)))))))));
14   forwards to if null(localErrors) then @fwrd
15     else errorStmt(localErrors);     }
```

Figure 14. An alternative version of Figure 8 using an error production instead of overriding the equation for `errors`. A forward production attribute is used to unconditionally decorate the translation when forwarding conditionally.

6.3 Forward production attributes

Overriding attributes on forwarding productions with values differing from those on the forward tree can lead to unexpected behavior when composing independent extensions, a problem known as *interference* [13]. To avoid this, *error productions* are included in host language specifications such as ableC, which can optionally be forwarded to instead of writing an override equation. However, this pattern is incompatible with computing inherited attributes through forwarding, as the forward tree may not always be decorated.

An alternative is to use a local *forward production attribute*, as seen on line 8 of Figure 14. This allows one to specify one forward tree for unconditionally supplying context, but potentially forward synthesized attributes to a different tree. We identified 49 places in the 10 ableC extensions evaluated in Section 4 where this feature would be useful.

7 Related Work

7.1 Attribute grammars

We added tree and contextual-information sharing to the Silver [25] attribute grammar system because the notion of forwarding makes the problem of sharing an interesting one. But there are other well-used AG systems that could have been used. Kiama [21] is a Scala library that also has a notion of forwarding. Similarly, JastAdd [3] has a notion of tree-rewriting [23] that is integrated into its use of reference [19] and circular [6] attributes. This may also be an interesting candidate for tree sharing to save re-computation of attributes. Silver has strategy attributes [18] that allow

one to write Stratego-style strategies to control the application or rewrite-rules [28]. The tree-sharing discussed here may be applicable in single-pass, bottom-up traversals since these are similar to what happens with translation attributes (Section 3.4). But it is not clear how well sharing can be incorporated into the more sophisticated strategies, and their ensuring traversal patterns, since it is unclear how one maintains the uniqueness requirements of the tree-sharing operator. This is certainly an area worth further study.

7.2 Tree sharing

The sharing of trees is a common practice in programming language tools. One influential example is the ATerm (Annotated Terms) system [24] for automatically sharing the representation of trees; it provides maximal sharing. Tree construction specifications will reuse existing trees if they already exist in the current collection of syntax trees. Kiama also provides an interesting notion of tree sharing in which the same tree can be decorated with two different values for the same set of attributes, an approach termed "respecting your parents" [22]. Here, the attribute values are stored separately from the tree in (unshared) attributions; they consist of a map from unique identifiers of tree nodes to attribute values. This is similar to Silver's distinguishing terms and decorated trees. In these works the aim of sharing is to represent trees more efficiently, and not to reuse or simplify the specification of computations.

Intentional Programming [20] is the most closely related work to our since forwarding was originally implemented in that system. It showed how extensions could specialize synthesized attributes, there called *questions*. But it automatically shared sub-trees under the forwarding and forwarded-to trees [27] and did not allow the specialization of inherited attributes, thus denying the language engineer the freedom to make these choices.

7.3 Attribute grammar flow analysis

Silver focuses on independent extensions so that a programmer can pick the ones they desire for their task at hand. Thus the modular well-definedness analysis [12] is used to ensure that the composition of these extensions will, in fact, work since the programmer is not in a position to debug or modify extension specifications; thus the extensions to this work in Section 5. Most related to our extensions is Boyland's analyses on remote attribute grammars [1]. That work analyses remote attributes and uses a notion of *fibers* to track dependencies not only on a remote node in the syntax tree but also the attributes that decorate it. This is similar to our extension of flow-types for synthesized attributes to also include the inherited attributes on a translation attribute on which it depends. Boyland's analysis was not a modular one and thus not directly applicable in our setting.

8 Future Work and Conclusion

8.1 Utilizing context supplied before forwarding

There are still some shortcomings with the approach to operator overloading in ableC proposed in Section 4. Every overloaded operator has both a forwarding production, and a non-forwarding default implementation production that is typically only ever constructed by its overloaded counterpart. Both productions must specify all the inherited equations needed for type checking, however with sharing, the equations on the non-forwarding production are never used.

To avoid specifying these equations multiple times, a solution is to identify productions like `intNeg` or `boolNeg` as *dispatch implementations* that can only be constructed as the forwarded-to tree of some specific *dispatching* production(s) like `neg`. The flow analysis from Section 5.2 can then be extended to consider any equations in the forwarding production as being supplied to the corresponding children in the implementation.

8.2 Data nonterminals

Sometimes nonterminals are used to represent data structures, such as optional values of type `Maybe` or an environment, that are never decorated with inherited attributes. Always automatically decorating children and locals of these types is inefficient and requires extra calls to `new`; instead we would like to mark them as *data nonterminals* that are never `Decorated`.

8.3 In conclusion

This paper introduces a new operator `@`, that permits tree-sharing without limiting extensibility. This allows language engineers to control when trees, and the specification of their contextual information, are to be shared or not. The examples given in Section 3 and the results of the evaluation in ableC indicate that tree sharing is the more common choice. But there are cases, especially when extensions are unlikely to be nested, when duplicating the tree is the right choice. This occurs in the ableC-Prolog extension discussed above, where host-language expressions appear in relations (*e.g.* in a numeric comparison goal), as well as in their translation. Since the generated code is complex, knowing *e.g.* what contextual information for a live-variable analysis is to be provided to expressions in a forwarding Prolog construct would be very difficult indeed. Here the right choice is to not share the child trees. Thus, it is important that language engineers have the freedom to make the appropriate choice, to share or not, and the new tree-sharing operator `@` provides that flexibility.

Acknowledgments

This work is partially supported by the National Science Foundation (NSF) under Grant Nos. 2123987.

Table 1. Information about the ableC extensions evaluated for the potential use of tree sharing.

Extension name	Description	Static	Dynamic	Translation attributes	Non-sharable	Forward prod attributes
condition-tables	Concise syntax for boolean expressions	7				1
closure	Lambda functions that capture scoped variables		2			5
algebraic-data-types	Algebraic data types implemented as tagged unions, with pattern matching	3		6		3
templating	C++-inspired templated function and type declarations		2			3
string	More efficient string representation type, overloadable operators for stringifying and pretty-printing various types	13				9
vector	Array-backed list data structure, with overloads for +, ==, and other operators					14
constructor	Syntax for constructing and deconstructing values, overloadable by other extensions such as vector	1	1			5
unification	Unification variable reference type and overloaded unify operator	5		3		3
prolog	Prolog-inspired logic programming relations and queries	4		6	15	3
rewriting	Stratego-inspired strategic term rewriting	7				4
Total	74 potential instances of sharing	39	5	15	15	49

A Evaluation of ableC extensions

Table 1 provides further details on the evaluation of opportunities for tree sharing in 10 representative ableC extensions. For each extension, from left to right is given

- The number of instances where tree sharing could be used in a statically-determined forward tree, supplying context and avoiding some explicit inherited equations;
- The number of instances where tree sharing could be used in a dynamic forward tree, where explicit inherited equations are still needed;
- The number of instances where tree sharing could be used in a translation attribute equation;
- The number of uses of `decExpr`-style wrapper productions that could not be replaced with the new tree sharing mechanisms, due to appearing in a non-unique context such as a higher-order inherited attribute equation;
- The number of productions in the grammar that would require a forward production attribute to enable static tree sharing.

The updated sources of these extensions can be found at https://github.com/melt-umn/ableC-<*extension-name*>; the versions evaluated here are archived at https://doi.org/10.13020/badh-qf44.

References

[1] John Tang Boyland. 2005. Remote attribute grammars. *J. ACM* 52, 4 (2005), 627–687. https://doi.org/10.1145/1082036.1082042

[2] Russel Cox, Tom Bergany, Austin Clements, Frans Kaashoek, and Eddie Kohlery. 2008. Xoc, an Extension-Oriented Compiler for Systems Programming. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 244–254. https://doi.org/10.1145/1353534.1346312

[3] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd extensible Java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications (OOPSLA)*. ACM, 1–18. https://doi.org/10.1145/1297027.1297029

[4] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming* 69 (December 2007), 14–26. Issue 1-3. https://doi.org/10.1016/j.scico.2007.02.003

[5] Sebastian Erdweg, Tillmann Rendel, Christian Kastner, and Klaus Ostermann. 2011. SugarJ: Library-based Syntactic Language Extensibility. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)*. ACM, 391–406. https://doi.org/10.1145/2048066.2048099

[6] R. Farrow. 1986. Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars. *ACM SIGPLAN Notices* 21, 7 (1986). https://doi.org/10.1145/13310.13320

[7] Görel Hedin. 2000. Reference Attribute Grammars. *Informatica* 24, 3 (2000), 301–317.

[8] T. Johnsson. 1987. Attribute grammars as a functional programming paradigm. In *Proc. of Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science, Vol. 274)*. Springer-Verlag, 154–173. https://doi.org/10.1007/3-540-18317-5_10

[9] Ted Kaminski. 2017. *Reliably Composable Language Extensions.* Ph. D. Dissertation. University of Minnesota, Minneapolis, Minnesota, USA. http://hdl.handle.net/11299/188954

[10] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. *Reliable and automatic composition of language extensions to C — Supplemental Material.* Technical Report 17-009. University of Minnesota, Department of Computer Science and Engineering. Available at https://hdl.handle.net/11299/216011.

[11] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. Reliable and Automatic Composition of Language Extensions to C: The ableC Extensible Language Framework. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 98 (Oct. 2017), 29 pages. https://doi.org/10.1145/3138224

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

[13] Ted Kaminski and Eric Van Wyk. 2017. Ensuring Non-interference of Composable Language Extensions. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE)* (Vancouver, Canada). ACM, 163–174. https://doi.org/10.1145/3136014.3136023

[14] Uwe Kastens. 1980. Ordered attributed grammars. *Acta Informatica* 13 (1980), 229–256. Issue 3. https://doi.org/10.1007/BF00288644

[15] Donald E. Knuth. 1968. Semantics of Context-free Languages. *Mathematical Systems Theory* 2, 2 (1968), 127–145. https://doi.org/10.1007/BF01692511 Corrections in **5**(1971) pp. 95–96.

[16] Donald E. Knuth. 1971. Semantics of Context-free Languages: Correction. *Mathematical Systems Theory* 5, 2 (1971), 95–96. https://doi.org/10.1007/BF01702865 Corrections to [15].

[17] Lucas Kramer and Eric Van Wyk. 2019. Parallel Nondeterministic Programming as a Language Extension to C (Short Paper). In *Proceedings of the International Conference on Generative Programming: Concepts & Experience (GPCE)* (Athens, Greece). ACM, 20–26. https://doi.org/10.1145/3357765.3359524

[18] Lucas Kramer and Eric Van Wyk. 2020. Strategic Tree Rewriting in Attribute Grammars. In *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering (SLE)* (Virtual, USA). 210–229. https://doi.org/10.1145/3426425.3426943

[19] Eva Magnusson and Görel Hedin. 2007. Circular reference attributed grammars - their evaluation and applications. *Science of Computer Programming* 68, 1 (2007), 21–37. https://doi.org/10.1016/j.scico.2005.06.005

[20] Charles Simonyi, Magnus Christerson, and Shane Clifford. 2006. Intentional software. *SIGPLAN Notices* 41, 10 (2006), 451–464. https://doi.org/10.1145/1167515.1167511

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

[22] Anthony M. Sloane, Matthew Roberts, and Leonard G. C. Hamey. 2014. Respect Your Parents: How Attribution and Rewriting Can Get Along. In *Software Language Engineering (Lecture Notes in Computer Science, Vol. 8706)*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer, 191–210. https://doi.org/10.1007/978-3-319-11245-9_11

[23] Emma Söderberg and Görel Hedin. 2015. Declarative rewriting through circular nonterminal attributes. *Computer Languages, Systems & Structures* 44 (2015), 3 – 23. https://doi.org/10.1016/j.cl.2015.08.008

[24] Mark G.J. van den Brand and Paul Klint. 2007. ATerms for manipulation and exchange of structured data: It's all about sharing. *Information and Software Technology* 49, 1 (2007), 55–64. https://doi.org/10.1016/j.infsof.2006.08.009

[25] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming* 75, 1–2 (January 2010), 39–54. https://doi.org/10.1016/j.scico.2009.07.004

[26] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. 2002. Forwarding in Attribute Grammars for Modular Language Design. In *Proceedings of the 11th Conference on Compiler Construction (CC) (Lecture Notes in Computer Science, Vol. 2304)*. Springer-Verlag, 128–142. https://doi.org/10.1007/3-540-45937-5_11

[27] E. Van Wyk, O. de Moor, G. Sittampalam, I. Sanabria-Piretti, K. Backhouse, and P. Kwiatkowski. 2001. *Intentional Programming: a Host of Language Features.* Technical Report PRG-RR-01-21. Computing Laboratory, University of Oxford.

[28] Eelco Visser. 2001. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA'01) (Lecture Notes in Computer Science, Vol. 2051)*, A. Middeldorp (Ed.). Springer-Verlag, 357–361. https://doi.org/10.1007/3-540-45127-7_27

[29] Harold H. Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. 1989. Higher Order Attribute Grammars. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 131–145. https://doi.org/10.1145/73141.74830

Received 2023-07-07; accepted 2023-09-01