Strategic Tree Rewriting in Attribute Grammars

Lucas Kramer
krame505@umn.edu
Department of Computer Science & Engineering
University of Minnesota
Minneapolis, MN, USA

Eric Van Wyk
evw@umn.edu
Department of Computer Science & Engineering
University of Minnesota
Minneapolis, MN, USA

Abstract

This paper presents strategy attributes, a seamless integration of strategic term rewriting into attribute grammars. Strategy attributes are specified using rewrite rules with strategies that control their application. The rules can reference contextual information held in attributes on the trees being rewritten. This use of attributes leads to rewriting on decorated trees instead of undecorated terms. During rewriting, attributes are (lazily) computed on new trees to ensure they are correct with respect to their defining equations. Attributes and strategic rewriting can each be used where most appropriate, thus avoiding the cumbersome aspects of each.

Strategy attributes are essentially higher-order attributes for which the defining equations are automatically generated from the attributes' strategy expressions. They are thus compatible with other attribute grammar features such as reference attributes, forwarding, and attribute flow analyses for well-definedness. A conservative static analysis checks if a strategy is intended to always succeed or to be partial, thus simplifying its use and optimizing its translation. Strategy attributes are demonstrated in the optimization of a simple expression language, evaluation of the lambda calculus, and optimization of strategy attribute translations.

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

Keywords: attribute grammars, term rewriting

ACM Reference Format:
Lucas Kramer and Eric Van Wyk. 2020. Strategic Tree Rewriting in Attribute Grammars. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3426425.3426943

SLE '20, November 16–17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8176-5/20/11…$15.00
https://doi.org/10.1145/3426425.3426943

1 Introduction and Motivation

Attribute grammars and strategic term rewriting are two long-standing techniques for specifying language semantics and transformations. Attribute grammars were invented by Knuth [24]. Since then there have been many extensions to the formalism [5, 12, 14, 39, 42] and many attribute grammar systems [2, 11, 13, 16, 20, 28, 30, 36, 40] have been developed. Term rewriting has a longer history [1, 9] in logic and languages, but our interest here is in strategic term rewriting in which strategies control the application of a set of rewrite rules. Strategies, in various forms, have been popularized by systems such as ASF+SDF [6], TOM [3], TXL [8], and of course, Stratego [41]. Both attribute grammars and term rewriting can be used for a wide range of applications, but each is most well-suited to different types of problems. The work presented here integrates strategic term rewriting into attribute grammars so that rewriting can be carried out over attribute-decorated trees instead of plain terms. This allows the strengths of each approach to be leveraged in solving a single problem. We call this *tree*-rewriting instead of *term*-rewriting to emphasize the point that there are attributes on syntax trees that can be used in the process, whereas terms are not associated with attributes.

In attribute grammars (AGs), syntax trees are decorated with semantic attributes, defined by equations associated with the productions in the grammar of the language. Specifying static analysis tasks such as name binding, type checking, and error reporting are common tasks for which this formalism is rather useful. However, AG specifications are often rather verbose when performing program transformations that modify only a small number of constructs in the language since equations must be written for all constructs. Transformation tasks are often more conveniently specified using strategic term rewriting approaches. Strategies specify how a set of rewrite rules, which are often not confluent, are applied over a term. This control lets one specify how the rewriting process traverses the tree and can be used to ensure the intended application order of the rules and also to provide speedups over non-strategic rewriting. However dealing with contextual information, such as a typing environment, is less straightforward than in AGs. In Stratego, for example, one may dynamically create new rewrite rules to track program variables and their types.

Traditionally attribute grammars and term rewriting systems have been separate tools or systems that are not easily

```
1  synthesized attribute
2      defs::[Pair<String Maybe<Expr>>];
3  inherited attribute
4      env::[Pair<String Maybe<Expr>>];
5  synthesized attribute freeVars::[String];
6  inherited attribute usedVars::[String];
7
8  nonterminal Decls
9    with defs, env, freeVars, usedVars;
10 production seq
11 ds::Decls ::= d1::Decls d2::Decls
12 { ds.defs = d1.defs ++ d2.defs;
13   d1.env = top.env;
14   d2.env = d1.defs ++ top.env; }
15 production empty Decls ::=
16 production decl  Decls ::= String Expr
17
18 nonterminal Expr  with env, freeVars;
19 production add   Expr ::= Expr Expr
20 production sub   Expr ::= Expr Expr
21 production neg   Expr ::= Expr
22 production const Expr ::= Integer
23 production letE  Expr ::= Decls Expr
24 production var   Expr ::= String
25 production app   Expr ::= String Exprs
```

Figure 1. Partial attribute grammar for a simple language.

used jointly. There are some exceptions, however. Kiama [36] supports strategic rewriting and attribution, but the processes are seen as essentially separate. Attributes can be used during rewriting, but they are not re-calculated on new trees constructed during a rewrite and this limits how effectively contextual information in attributes can be used. We have also extended the Silver [40] AG system with a mechanism [27] for rewriting on undecorated terms based on reflection [26]. This is similar to Kiama in that attributes can only be used in rules by explicitly providing values for any required inherited attributes. Rewriting in the JastAdd [11] AG system and forwarding in Silver can make full use of attributes, but forwarding is limited to a single one-step transformation and rewriting in JastAdd does not support strategies. Both are also primarily used to de-sugar language extensions and not to perform, for example, optimizing transformations. Our aim is to provide a seamless integration of the two paradigms.

Motivating Example: To motivate this work, consider a portion of a Silver specification for a small expression language in Figure 1. Silver [40] is an extensible AG system and our strategic tree rewriting integration is done as an extension to it. A program in this language is a sequence of declarations, represented by the (abstract) grammar containing the `Decls` nonterminal and associated productions on lines 8—16. `Decls` can be a sequence of declarations (line 10), an empty declaration (line 15), binding a name to an expression (line 16), or a function declaration (not shown.) Lines 18—25 specify the `Expr` nonterminal for expressions, with productions for arithmetic, let-expressions, variable references, and named function application. These nonterminals are decorated `with` the inherited attribute `env`, a list of pairs mapping strings to optional (`Maybe`) expressions (to be explained shortly), and the synthesized attribute `freeVars`, the list of free variables in the expression. The attribute `defs` collects declarations on let-expressions to populate the environment and `usedVars` is the list of names used in the scope of a declaration. The production `seq` on lines 10—14 shows how lists of definitions are collected and passed down the tree in the inherited `env` attribute. Note that `++` is list concatenation and that nonterminals in the production signature can be named in order to reference their attributes.

The motivating example here optimizes expressions using constants as expressed by the following rewrite rules:

$$\texttt{add}(e, \texttt{const}(0)) \rightarrow e \tag{1}$$

$$\texttt{add}(\texttt{const}(0), e) \rightarrow e \tag{2}$$

$$\texttt{add}(\texttt{const}(a), \texttt{const}(b)) \rightarrow \texttt{const}(a + b) \tag{3}$$

$$\texttt{sub}(e_1, e_2) \rightarrow \texttt{add}(e_1, \texttt{neg}(e_2)) \tag{4}$$

$$\texttt{neg}(\texttt{neg}(e)) \rightarrow e \tag{5}$$

$$\texttt{neg}(\texttt{const}(a)) \rightarrow \texttt{const}(-a) \tag{6}$$

$$\texttt{var}(id) \mid (id, \texttt{just}(e)) \in env \rightarrow e \tag{7}$$

The first 6 rules require no use of contextual information, but rule 7 does. The environment *env* indicates if an identifier should be inlined in this optimization by mapping it to its defining expression (in `decl` on line 16) wrapped up in the `just` constructor for type `Maybe`. Since let-expressions support name shadowing, all bound names must appear in the environment. Those identifiers that will not be inlined are paired with a `nothing()` value. In Section 3 we will see the attribute grammar specification that determines if an identifier is inlined; this is done when the bound expression has no free variables or the identifier is used just once in its bound scope, which is determined using attributes. The overall transformation is driven by strategies to control the application of the rules above. The inlining of some expressions for identifiers will enable the inlining of others in a later step as the number of identifier uses decreases. This decrease is realized in computing these attributes on new trees as they are created during the strategic rewrite. These strategies must also, for example, ensure that after rule 4 produces a term, the result is further optimized when possible. Specifying rules and strategies like these in attribute grammars is rather awkward and more verbose than a strategy based specification; the result from applying a rule must be

re-decorated with attributes to compute the next step in the optimization, and must also be able to stop when applying the optimization does not further simplify the term. This example demonstrates the benefit of using both attributes and strategies in the same task as enabled by strategy attributes.

Strategy attributes: Strategy attributes provide a seamless integration of attribute grammars and term rewriting to address the shortcomings of both as discussed above. Rewriting strategies are expressed in a domain-specific language based on the strategy expressions in Stratego and similar systems and realized as strategy attributes. These attributes are simply higher-order attributes [42] that store the result of applying the strategy to the tree on which the attribute occurs. One does not write equations for these attributes; they are automatically generated from the strategy expressions.

For example, we define a strategy attribute named `optimize` to perform the optimizations from rules 1–6 as:

```
strategy attribute optimize = all(optimize)
    <* ((optimizeStep <* optimize) <+ id);
attribute optimize occurs on Expr;
```

This recursive strategy expression applies itself to `all` child trees, and then (using the `<*` sequence combinator) applies `optimizeStep` and (if this succeeds) itself to the current tree, or (`<+` is the choice combinator) succeeds with the identity transformation; note that `optimizeStep` is the strategy attribute that realizes rules 1–6. There is no operator to apply a strategy, to get the result one simply accesses the attribute in the same manner as for traditional higher-order attributes.

As will be demonstrated in Section 3.3, a critical aspect of this seamless integration is the capability of rewrite rules, as constructs in strategy expressions, to reference traditional attributes on the decorated trees being rewritten. No explicit redecoration directives are needed. This allows intermediate trees generated during a rewriting to be correctly decorated with the appropriate inherited attributes.

Note that the `optimize` strategy above will always produce a tree. We refer to these as *total* strategy attributes and they have the same type as the nonterminal they decorate, in the case above the type of `optimize` on an `Expr` is `Expr`. Some strategies may fail, for example `optimizeStep` will fail when applied to an addition term in which neither child is a constant; these are *partial* strategies and are labeled by a **`partial`** keyword modifier. When these occur on a nonterminal `X`, the attribute type is `Maybe<X>`. An access must explicitly check if the rewrite succeeded. A conservative static analysis checks that total strategies will succeed, if they terminate (termination is not part of the totality check.) This simplifies the use of total strategies in that checks for failure are not needed.

Contributions: This paper's primary contributions are:

- a seamless integration of strategic rewriting of syntax trees and attribute grammars. (§ 3)
- a conservative totality (not including termination) analysis of strategy attributes to simplify their use and support their optimization. (§ 3.1)
- a translation of strategic tree rewriting specifications into traditional higher-order attributes and equations that implement them (§ 4) thus ensuring that strategy attributes are compatible with other attribute grammar features such as reference attributes [5, 14], forwarding [39], and well-definedness analyses [24].
- the implementation of strategy attributes as a modular composable language extension to Silver. (§ 4.3)

Section 2 provides background on AGs and strategic term rewriting. Section 5 presents several demonstrations of integrating attributes and strategies in the rewriting process, including expression inlining (§ 3.3), evaluation of the λ-calculus (§ 5.1), optimized regular expression matching with Brzozowski derivative (§ 5.2), for-loop normalization showing integration with Silver and forwarding, (§ 5.3), and optimizing strategy expressions in the strategy attribute translation process (§ 5.4). Section 6 presents related work and Section 7 discusses some limitations of strategy attributes and draws some conclusions. Silver version 0.4.4 [1] [40] is the attribute grammar system used in this paper. Specifications of the demonstrations listed above, along with the software artifact for running them, are also freely available. [2]

2 Background

2.1 Attribute Grammars

As sketched in Section 1, attribute grammars [24, 33] are a formalism for defining language semantics, such as a static analysis for type checking or a program translation. Formally, an attribute grammar is a tuple $AG = \langle G, A, EN, O, EQ \rangle$ in which G is a context free grammar, A is a set of attributes that decorate nodes in trees in the language of G, and the relationship O indicates which attributes occur on which nonterminal symbols in G. A grammar $G = \langle NT, T, P \rangle$ consists of NT, a finite set of nonterminal symbols, T, a finite set of terminal symbols, and P, a finite set of productions. T often includes some primitive types such as string and numbers. Productions in P have a left-hand side nonterminal NT_0 and 0 or more right-hand side symbols in $X_i \in NT \cup T$. These symbols are labeled and have the form $n :: NT_0 ::= n_1 :: X_1 \ldots n_n :: X_n$. The labels, *e.g.* n, n_i, are used to reference specific tree nodes, as seen in Figure 1 in the seq production. EN is a typing environment containing attribute types and production signatures. $EQ = \cup_{p \in P} EQ_p$ is a set of equations, associated with specified productions, that define the values of attributes on tree nodes. In Silver equations are written with the production declaration

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

```
SE ::= rule on Ty of Rules end                    (rule)
     | SE <* SE | SE <+ SE                        (sequence, choice)
     | all(SE) | some(SE) | one(SE)               (generic traversals)
     | fail | id | Ident                          (strategy reference)
     | Ident(SE, ..., SE)                         (congruence traversal)
     | rec Ident -> SE                            (recursive strategy)
```

Figure 2. The syntax of strategy expressions.

(lines 10—14) or with an `aspect` production in order to associate equations with an existing production, for example, lines 12—17 in Figure 6 which gives equations for `usedVars` on the `seq` production.

Attributes can hold values of various types beyond simple strings and integers. Strategy attributes translate into higher-order attributes [42] which contain (yet undecorated) syntax trees. Reference [14] and remote [5] attributes allow decorated syntax trees to be passed around as attribute values; these are sometimes thought of as references or pointers to distant nodes in the syntax tree. Syntax trees that are not children of productions can also be decorated, *i.e.* provided with inherited attributes. These can be `local` higher-order attributes associated with a production, or done in an expression using the `decorate` t `with` $\{i_1 = e_1; \ldots; i_n = e_n\}$ form to provide tree t with values for inherited attribute i_j.

A variety of approaches can be used to evaluate equations in a satisfiable order, including modern demand-driven approach [15] as used in Silver and JastAdd and the ordered approach [21] that statically schedules equations for evaluation. The demand-driven approach works well for strategy attributes because it ensures that no unnecessary trees or attributes are calculated.

2.2 Strategic Term Rewriting

Term-rewriting is another formalism useful for specifying language translations and transformations. These specifications also include context free grammars to define the language being transformed, or the source and target languages in a translation application. As seen in Section 1, a rewrite rule consists of a pattern with free variables on the left, and a term that uses those free variables on the right. A *redex* is a part of a term that matches the pattern, it is replaced by the instantiated term on the right, called the *contractum*. Simple term rewriting searches for redexes, replacing them with their contractum, and terminates when no more redexes can be found. A set of rules is confluent if the result of rewriting is independent of the order and location in which the rules are applied. Specifying rewrite rules to be confluent is difficult and can result in complex rules that obscure the intention of the application. To address this, many [3, 4, 41] have proposed using *strategies* as a means of controlling the application of a set of rewrite rules so that the outcome for non-confluent rules is determined.

Figure 2 has the grammar for the language of strategies (*SE*) used in this paper. These are all based on strategy combinators in Stratego and found in other systems as well, with only small changes in syntax. A strategy defines a partial function on terms; it may succeed and produce a new term, or fail to do so. The fundamental strategy (rule) is a collection of rewrite rules (*Rules*) over a specified nonterminal type (*Ty*). Strategies can be combined to be applied in sequence so that if the first strategy produces a term then the second strategy is applied to it. If the first strategy fails, then the sequence fails. The choice combinator tries the first strategy and if it succeeds returns that result, otherwise it tries the second. The generic traversal strategies are applied to the children of a term. If the strategy *s* succeeds on all children of a term, then `all(s)` succeeds by reconstructing the tree with the new resulting children. The `some(s)` strategy is similar and succeeds if *s* succeeds on at least one child, and `one(s)` applies *s* to the child terms, stopping on the first success. Two primitive strategies are `fail`, which always fails, and `id`, which succeeds by returning the original term. These are commonly composed to create new strategies:

```
try(s) = s <+ id                                          (8)
bottomUp(s) = all(bottomUp(s)) <* s                       (9)
innermost(s) = bottomUp(try(s <* innermost(s)))           (10)
```

Here, `try` converts a strategy `s` that may fail into one that will succeed, by applying `s` and if it fails instead succeeding with `id`. The `bottomUp` combinator applies a strategy from the bottom to the top of the term, but fails if the input strategy fails at any location in the term. `innermost` uses `bottomUp` to repeatedly apply `s` with preference for the innermost unreduced sub-terms, until `s` fails everywhere.

Strategies may be named, and may be referred to by other strategies. A congruence traversal strategy is used to target a term built by a specific named production, and fail on all others. This strategy is constructed by that name and a particular strategy for each child in the term. An example of this is discussed in Section 3. The last strategy in Figure 2 is used to define recursive strategies without naming them in a top-level declaration, a bit like lambda-expressions. This is often used in defining new strategy combinators (like `innermost`) where strategy declarations would be cumbersome.

3 Design and Use of Strategy Attributes

As laid out in Section 1, strategies are specified and used as *strategy attributes*. These are essentially higher-order attributes for which the defining equations are generated from strategy expressions. The specification of the `optimizeStep` strategy implementing rules 1—6 over the language in Figure 1 is given in Figure 3 (the `optimize` strategy is repeated here as well.) `optimize` is a partial strategy that may fail, so its type on `Expr` nonterminals is `Maybe<Expr>`, whereas `optimize` is total, so its type on a nonterminal it decorates is just that

```
1  partial strategy attribute optimizeStep =
2    rule on Expr of
3    | add(e, const(0)) -> e
4    | add(const(0), e) -> e
5    | add(const(a), const(b)) -> const(a + b)
6    | sub(e1, e2) -> add(e1, neg(e2))
7    | neg(neg(e)) -> e
8    | neg(const(a)) -> const(-a)
9    end  occurs on Expr;
10 strategy attribute optimize = all(optimize)
11     <* ((optimizeStep <* optimize) <+ id)
12   occurs on Expr, Exprs, Decls;
13 propagate optimizeStep on Expr;
14 propagate optimize on Expr, Exprs, Decls;
```

Figure 3. A specification of strategy attributes implementing rewrite rules 1—6; rule 7 is implemented later in Figure 6.

$$\textsc{Id}\ \frac{}{\Gamma \vdash \texttt{id}\ \mathit{total}} \qquad \textsc{Seq}\ \frac{\Gamma \vdash s_1\ \mathit{total} \qquad \Gamma \vdash s_2\ \mathit{total}}{\Gamma \vdash s_1\ \texttt{<*}\ s_2\ \mathit{total}}$$

$$\textsc{ChoiceL}\ \frac{\Gamma \vdash s_1\ \mathit{total}}{\Gamma \vdash s_1\ \texttt{<+}\ s_2\ \mathit{total}} \qquad \textsc{ChoiceR}\ \frac{\Gamma \vdash s_2\ \mathit{total}}{\Gamma \vdash s_1\ \texttt{<+}\ s_2\ \mathit{total}}$$

$$\textsc{All}\ \frac{\Gamma \vdash s\ \mathit{total}}{\Gamma \vdash \texttt{all}(s)\ \mathit{total}} \qquad \textsc{Ref}\ \frac{n \in \Gamma}{\Gamma \vdash n\ \mathit{total}} \qquad \textsc{Rec}\ \frac{\Gamma \cup \{n\} \vdash s\ \mathit{total}}{\Gamma \vdash \texttt{rec}\ n\ \texttt{->}\ s\ \mathit{total}}$$

Figure 4. Inference rules for totality checking of strategy expressions with respect to an environment Γ containing the names of all total strategy attributes.

nonterminal; for `Expr` it is just `Expr`. Strategy attributes not labeled as partial are considered to be total. [3]

The right side of a strategy attribute definition is a domain-specific language of strategy expressions, describing rewrite rules and traversal orderings, as explained in Section 2.2. This is compiled into attribute equations that implement the strategic rewrite for each production on which the attribute is propagated. The generation of these equations is triggered by a **`propagate`** declaration, as seen on line 13 and 14. This is an overloaded construct in Silver that allows attribute equations to be generated that will propagate attribute values over the syntax tree. The **`propagate`** declaration for `optimize`, discussed in more detail in Section 4, will generate `aspect` productions for all productions with the listed nonterminals on their right hand side (*e.g.* `add`, `sub`, `seq`, and other in Figure 1). These will contain the generated equations defining the synthesized attribute `optimize`.

3.1 Totality Analysis of Core Strategy Combinators

A (conservative) static analysis is performed on strategy expressions to determine if they are partial or total. A warning is emitted when a strategy specified to be total is defined with a partial strategy expression, and a run-time error is raised if such an erroneous strategy fails. This analysis is presented as a set of inference rules in Figure 4 over the core strategy combinators shown in Figure 2. A strategy s is total (written $\Gamma \vdash s\ \mathit{total}$) if such a derivation is possible from the rules. Γ is an environment containing the names of all strategy attributes declared to be total, as well as in-scope **`rec`** variables. The `id` strategy always succeeds with the current tree and the rule Id provides a derivation for this. On the other hand, `fail` always fails; there is no derivation of $\Gamma \vdash \texttt{fail}\ \mathit{total}$ for any Γ.

The **`rule`** strategy (such as `optimizeStep`) is a sequence of patterns on a nonterminal and will succeed when one of its patterns matches the current tree, or will fail otherwise. Because a **`rule`** is defined for a specific nonterminal, and strategies can be applied to (occur on) multiple nonterminals we consider rules to always be partial.

Sequence (`<*`), such as seen on line 11 of Figure 3, applies its right operand to the term resulting from applying its left operand to the current tree. Sequence succeeds when both operands succeed and thus is total only when its arguments are both total, as indicated in Seq. Choice (`<+`), such as seen on line 11 of Figure 3, first applies its left operand to the current tree, and if unsuccessful attempts the same for its right operand. Thus choice is total when either the left (ChoiceL) or right operand (ChoiceR) is total. Note that this means the right operand is ignored when the left is total.

One can also reference the names of other strategy attributes that should be computed on the current tree, as is done for `optimizeStep` and `optimize` on line 11 of Figure 3. Since strategy attributes are translated into higher-order attributes one can reference any higher-order attribute of the correct type. Such a reference is total only if the referenced strategy attribute was declared as total (Ref). This can allow for the definition of recursive strategies, or allow for rules to be factored out into separate strategy attributes. When a strategy attribute (such as `optimize`) references a partial strategy attribute (such as `optimizeStep`), the reference will be treated as failure for any nonterminals on which the referenced attribute does not occur (such as `Decls`). Conversely when a total strategy attribute is referenced, it is an error for the referenced attribute to not occur on all of the same nonterminals, as a reference to a total strategy attribute is itself total and thus has no way to fail.

[3] Note that *total* here means a tree will be produced if the strategy terminates. No guarantee of termination is to be assumed for attributes in this category. Similar terminology is used in Scala for its partial and total functions.

strategy	translation
`try(s)`	`s <+ id`
`repeat(s)`	`rec x -> try(s <* x)`
`bottomUp(s)`	`rec x -> all(x) <* s`
`topDown(s)`	`rec x -> s <* all(x)`
`downUp(s1, s2)`	`rec x -> s1 <* all(x) <* s2`
`allTopDown(s)`	`rec x -> s <+ all(x)`
`onceBottomUp(s)`	`rec x -> one(x) <+ s`
`innermost(s)`	`rec x -> bottomUp(try(s <* x))`

Figure 5. Some of the utility strategies and their translations, extending utility strategies in rules.

The generic traversal combinators `all`, `some` and `one` allow for the argument strategy to be applied across the children of the current tree, producing a new term with the same production wrapping the results. Any children on which the argument does not occur are ignored by the traversal and copied unchanged into the new term. The `all` strategy, such as on line 10 of Figure 3, succeeds if the argument was applied successfully to all the children, thus it is total if the argument strategy is total (All). The `some` strategy succeeds if the argument succeeds for at least one child; any failing children are left unchanged. However, as the strategy fails for any production with no children, it cannot be total for all nonterminals. The `one` strategy is quite similar to `some`, except that it works left to right and stops after the first succeeding child.

3.2 Utility Strategies

As discussed in Section 2.2, many common patterns exist in strategy expression specifications leading to the definition of new strategies composed from others, such as `try` and others in rules 8–10. A set of language extensions to the strategy expression language provides utilities for some of these patterns. Figure 5 shows some of these, along with their translation (using forwarding [39]). Note that the `optimize` strategy defined in Figure 3 could have been equivalently defined as `innermost(optimizeStep)`. The specification of many utility strategies are recursively defined, *e.g.*, the strategy `bottomUp(s)` is equivalent to `all(bottomUp(s)) <* s`. This presents a problem, as directly performing the translation in this way would result in an infinite strategy expression term being constructed, of the form

```
all(all(all(...) <* s) <* s) <* s
```

This can be avoided by using the **`rec`** strategy expression combinator: **`rec`** *x* `->` *body*. This binds a name *x* to the strategy expression *body*, such that *body* can refer to itself with the name *x*. This allows us to instead define `bottomUp(s)` as `rec x -> all(x) <* s`.

The addition of **`rec`** complicates the totality checking process, as the totality of a reference to a **`rec`** variable now depends on the totality of the enclosing **`rec`** expression. However this can be resolved by observing that the totality of a **`rec`** expression does not actually depend on the totality of references to its variable (See Rec). If for a particular **`rec`** expression one assumes that all references to its variable are total, and the analysis determines that the body is indeed total, then the correct assumption was made. On the other hand if the body is determined to be partial, then although the variable was incorrectly assumed to be total, correctly assuming it to be partial would not have affected the result of the analysis. Note that to correctly determine whether sub-expressions of a **`rec`** body are total, the analysis must still be repeated on the body with Γ updated according to the result of the first pass. Since the utility strategies are translated using forwarding, the totality analysis for them is automatically carried out on this translation.

3.3 Integration of Rewriting and Attributes

Strategy attributes integrate strategic rewriting into attribute grammars in a seamless way that lead to *tree* rewriting instead of *term* rewriting. The advantage of tree rewriting is that the rewriting process can use inherited and synthesized attributes in rewrite rules to determine where they can be applied and what they will produce. An important aspect of this is that, as new trees are created, they are given inherited attributes so that they may be re-decorated and those new attribute values may be used in further rewriting steps.

An example of this is shown in Figure 6, where strategy attributes are used to perform inlining of let declarations (rule 7 from Section 1) in addition to the previously described optimizations. The `freeVars` and `usedVars` attributes are used to determine if a declaration should be inlined (line 6); here we only wish to inline constant expressions (those with no free variables, line 6) and variables with at most one reference, to avoid duplicating computations (lines 7—8). If these conditions hold (`inline`) then the bound expression is put in the environment in a `just` constructor, otherwise `nothing()` is associated with the identifier. (This information reaches the inherited environment attribute `env` via the synthesized attribute `defs` that flows up the tree to the point where they are passed to the appropriate expressions, *e.g.* lines 10—14 in Figure 1.) The first rule of the `inlineStep` strategy (lines 20—21) then attempts to replace variable references with the corresponding expression by looking them up in the inherited environment. The rule is applied when the result of looking up `n` in the environment (`top.env`) finds `n` (the outer `just`) and the expression is to be inlined (the inner `just`). It is this inherited attribute access, `top.env`, that is important. Note that synthesized attributes can also be accessed in a rule. These will be calculated based on the proper inherited attributes that are provided during the rewriting process. The following rules (lines 22—29) remove bindings that are no longer referenced as a result of inlining.

```
1  aspect production letE
2  top::Expr ::= d::Decls e::Expr
3  { d.usedVars = e.freeVars; }
4  aspect production decl
5  top::Decls ::= id::String e::Expr
6  { local inline::Boolean = null(e.freeVars)
7      || length(filter( (==id),
8                        top.usedVars )) <= 1;
9    top.defs = [pair(id,
10     if inline then just(e) else nothing())];
11 }
12 aspect production seq
13 top::Decls ::= s1::Decls s2::Decls
14 { d1.usedVars = d2.freeVars ++
15     removeAll(map(fst, d2.defs),
16               top.usedVars);
17   d2.usedVars = top.usedVars; }
18 partial strategy attribute inlineStep =
19   rule on top::Expr of
20   | var(n) when lookup(n, top.env)
21         matches just(just(e)) -> e
22   | letE(empty(), e) -> e
23   end <+
24   rule on top::Decls of
25   | decl(id, e) when
26       !contains(id, top.usedVars) -> empty()
27   | seq(d, empty()) -> d
28   | seq(empty(), d) -> d
29   end
30   occurs on Expr, Decls;
31 strategy attribute optimizeInline = repeat(
32   onceBottomUp(optimizeStep <+ inlineStep))
33   occurs on Expr, Exprs, Decls;
34
35 propagate inlineStep on Expr, Decls;
36 propagate optimizeInline on Expr, Exprs,
37                             Decls;
```

Figure 6. Strategy attributes used in conjunction with synthesized and inherited attributes to simplify expressions as in Figure 3 and to perform an inlining optimization (rule 7).

Lines 31—33 define the overall strategy using `repeat` and `onceBottomUp` to control the application of the simpler stepping strategies for optimizing expressions (`optimizeStep`) and inlining declaration (`inlineStep`). The `onceBottomUp` strategy works from the bottom and left of a tree to find the first successful rewrite of `optimizeStep <+ inlineStep` and then does no more rewriting. This is repeated (`repeat`) until no more rewrites are made. The translation of `repeat` uses sequence (`<*`) and it, in `s1 <* s2`, will decorate the result of a successful application of `s1` with the inherited attributes of the original term in order to apply `s2`. This is inefficient since it traverses the entire tree for each step made. Care must be taken in designing such strategies to avoid excessive decoration of new trees when more efficient strategies exist. Note that strategies that do not use sequence, such as `allTopDown`, by definition constitute a single pass and will not result in any redecoration.

One might think to try `innermost` over these stepping strategies. Unfortunately, the use of `all` in its definition would cause both children of a `seq` or `letE` production to be rewritten simultaneously with their original inherited attribute values, leading to missed optimizations. For example, inlining `let a = 1 in a` would result in `let a = 1 in 1`, because the declaration gets the old value for `usedVars`, still containing `a`. Instead we must rewrite the first child, decorate the resulting term, and only then optimize the second child (which has now been decorated with an updated `env`.) The first child must then be rewritten once again using the `usedVars` arising from the second term. This can be achieved by combined use of the `repeat` and `onceBottomUp` strategies.

3.4 Congruence Traversal Operators

While correct, the implementation of `optimizeInline` in Figure 6 is rather inefficient, as the entire tree is re-traversed for every optimization step that is performed. It is still safe to optimize and inline within most productions in an innermost manner without redecorating the entire sub-tree between each step, but more careful control over the traversal and decoration of `seq` and `letT` is needed. This can be accomplished by using *congruence* traversal strategies, originally introduced by Stratego [41]. A congruence strategy is specified as the name of a production, applied to a list of strategy expressions corresponding to each production argument. When applied to the matching production, each argument strategy is applied to the corresponding child subtree of the current tree. If all child strategies succeed then the congruence succeeds with a new term consisting of the same production applied to each child result. Since they only succeed for a particular production, congruence traversal strategy expressions are always partial.

A more efficient version of `optimizeInline` that uses congruence traversal strategies is shown in Figure 7. The definition of this strategy mirrors the definition of `innermost`, except for the addition of special cases in handling traversals. Line 2 deals with `seq` declarations, consisting of 3 congruence sub-expressions chained together in sequence (`<*`). The first fails when the current term is not `seq`, and otherwise recursively applies `optimizeInline` to the left child while doing nothing on the right child. When this succeeds, the sequence operator re-decorates the resulting term with the same inherited attributes, and applies the second sub-expression to rewrite the right child. Since the tree has been re-decorated, the new inherited attributes resulting from rewriting the left child are automatically provided to the right child. This is repeated for a third time by the remaining sub-expression,

```
1  strategy attribute optimizeInline =
2    ((seq(optimizeInline, id) <* seq(id, optimizeInline) <* seq(optimizeInline, id)) <+
3     (letE(optimizeInline, id) <* letE(id, optimizeInline) <* letE(optimizeInline, id)) <+
4     all(optimizeInline)) <* try((optimizeStep <+ inlineStep) <* optimizeInline);
```

Figure 7. A more efficient implementation of the `optimizeInline` strategy from Figure 6 using congruence traversal strategies.

rewriting the left child again using new inherited attributes (`usedVars`) arising from the rewritten right child. `letE` productions are handled similarly by congruences on line 3. All other productions are handled by the `all` traversal combinator on line 4. Since `all(optimizeInline)` is total, the first half of this strategy is total as well. The remainder of the strategy is the same as `innermost`: if `optimizeStep` or `inlineStep` succeeds, then `optimizeInline` is repeated again on the result.

4 Implementation

Here we describe the translation of strategy attributes into higher-order attributes with equations for all the productions for nonterminals a strategy attribute is propagated on. Part of the translation of the `optimize` strategy from Figure 3 for the `Expr` nonterminals is shown in Figures 8 and 9. For clarity, some further optimizations have been performed; these will be explained in Section 5.4.

4.1 Basic Strategies

Recall that strategy expressions can be total or partial, thus the values we are operating on can either be nonterminals (*e.g.* the `optimize` strategy) or nonterminals wrapped in `Maybe` (*e.g.* `optimizeStep`). Consider the total `id` strategy expression; it is simply translated as a reference to the current tree (the label for the production's left-hand side nonterminal). Conversely, `fail` (a partial strategy) is translated as `nothing()`, which has type `Maybe<a>`. When a partial strategy attribute is defined with a total strategy expression, the translation of the expression can be made partial by wrapping it in `just`. However if a total strategy attribute is defined with a partial strategy expression (causing a compile-time warning to be emitted), a run-time error check is performed using pattern matching to unwrap the `Maybe` value.

Rule strategies are translated into Silver pattern matching `case` expressions. Consider the rule in the `optimizeStep` strategy in Figure 3; since it is partial it has type `Maybe<Expr>`, as it may either succeed with a new term or fail. This rule becomes a pattern match on the current tree that includes only the pattern rules corresponding to the production. This can be seen in the four case-clauses for the `add` production on lines 4–7 of Figure 9. The right side of each pattern rule (*e.g.*, line 5) is wrapped in `just` since the strategy is partial. A default pattern is added that returns `nothing()` when all of the rules fail to match. On productions where none of the rule's patterns match (such as `const`), the entire rule is guaranteed to fail and can be translated as `nothing()` (line 13.)

References to other attributes that occur on the nonterminal are simply translated by accessing the attribute on the current tree, as seen with the access of `top.optimizeStep` on line 10 of Figure 9. Partial strategy attributes that don't occur on the nonterminal, *e.g.* `optimizeStep` on `Decls`, may also be referenced; such references are translated as `nothing()`.

Sometimes translating a strategy expression requires the generation of additional "helper" strategy attributes. This is done by *lifting* sub-expressions of the strategy expression into additional strategy attribute declarations, that occur on and are translated for the same nonterminals as the original strategy attribute from which they were generated. This can be seen with the translation of sequence in Figure 8, where a portion of the `optimize` strategy expression has been lifted into a new strategy attribute `optimize_snd`. The resulting `propagate` declaration (line 4) is then translated to produce the equations in the lower part of the figure.

The application of a sequence (`<*`) first applies its left operand to the current tree, then applies its right operand to that result when the left operands succeeds. Since left operand is applied to the current tree, its translation may be computed as usual. However, the right operand is no longer being applied to the current tree. When the right operand is simply a reference to another attribute, we can access this attribute on the left result. But when the right operand is not a reference to an attribute, we must lift it to make it one.

This translation of sequence depends on the totality of its operands, as shown in Figure 10. For simplicity, let us consider the case when both operands are references to strategy attributes `s1` and `s2`, such that no lifting is required, and the nonterminal has a single inherited attribute `env`. When both strategies are total and inherited attributes are not considered, we could translate `s1 <* s2` as `top.s1.s2`. However, strategy attributes are computed on trees that have been decorated with inherited attributes, whereas the result of applying a strategy is an undecorated term. Thus we must explicitly decorate the left result `s1.top` before we can access the attribute `s2` corresponding to the right expression. When `s1` is total and `s2` is partial, we can still directly access `s2` on the decorated result of `top.s1`, and thus the translation for these cases is the same. However when `s1` is partial, its result is wrapped in a `Maybe` value; this requires a functor map or monadic bind operation in the translation. Note that both translations are given in a functional style using `fmap` or monadic bind (`>>=`), followed by the equivalent, easier-to-read expansion into a case expression.

```
strategy attribute optimize = all(optimize) <* optimize_snd  occurs on Expr, Exprs, Decls;
strategy attribute optimize_snd = (optimizeStep <* optimize) <+ id
    occurs on Expr, Exprs, Decls;
propagate optimize, optimize_snd on Expr, Exprs, Decls;
```

Figure 8. The lifted transformation of the `optimize` strategy from Figure 3.

```
synthesized attribute optimizeStep<a>::Maybe<a>;
attribute optimizeStep<Expr>, optimize<Expr>, optimize_snd<Expr> occurs on Expr;
aspect production add    top::Expr ::= e1::Expr e2::Expr
{ top.optimizeStep = case top of
    | add(e, const(0)) -> just(e)                    | add(const(0), e) -> just(e)
    | add(const(a), const(b)) -> just(const(a + b))  | _ -> nothing()
  end;
  top.optimize = decorate add(e1.optimize, e2.optimize) with {env = top.env;}.optimize_snd;
  top.optimize_snd = fromMaybe(top,
    fmap(\ a::Expr -> decorate a with { env = top.env; }.optimize, top.optimizeStep)); }

aspect production const  top::Expr ::= i::Integer
{ top.optimizeStep = nothing(); top.optimize = top.optimize_snd; top.optimize_snd = top; }
```

Figure 9. The (optimized) translation of the specifications in Figure 3 for the `add` and `const` productions.

s1	s2	s1 <* s2	Translation
T	T	T	`(decorate top.s1 with { env = top.env; }).s2`
T	P	P	`(decorate top.s1 with { env = top.env; }).s2`

P	T	P	`fmap(\ res -> decorate res with { env = top.env; }.s2, top.s1)`
			`= case top.s1 of just(res) -> just(decorate res with { env = top.env }.s2)`
			`| nothing() -> nothing() end`

P	P	P	`top.s1 >>= \ res -> decorate res with { env = top.env; }.s2`
			`= case top.s1 of just(res) -> decorate res with { env = top.env }.s2`
			`| nothing() -> nothing() end`

Figure 10. The translation of `s1 <* s2`, for various combinations of partial (P) and total (T) strategy attributes `s1` and `s2`. The nonterminal has a single inherited attribute, `env`.

s1	s2	s1 <+ s2	Translation
T	*	T	`top.s1`

P	T	T	`fromMaybe(top.s2, top.s1)`
			`= case top.s1 of`
			`| just(res) -> res`
			`| nothing() -> top.s2 end`

P	P	P	`top.s1 <> top.s2`
			`= case top.s1 of`
			`| just(res) -> just(res)`
			`| nothing() -> top.s2 end`

Figure 11. The translation of `s1 <+ s2`, for various combinations of partial (P) and total (T) strategy attributes `s1` and `s2`. The nonterminal has a single inherited attribute, `env`.

Choice is similar to sequence, except that both options are applied to the current tree, so lifting is never needed. Its translation for various combinations of partial and total strategies is shown in Figure 11. When `s1` is total, `s2` is ignored as the choice will always succeed with `s1`. Otherwise the results of `s1` and `s2` are combined using the standard library function `fromMaybe :: (a ::= a Maybe<a>)` or the monoid append operator `<>`. Again, for the latter two cases a compact and an equivalent expanded version are given.

4.2 Traversal Strategies

The `all` traversal strategy applies its argument strategy expression to all children of the current tree, and constructs a new term from the results using the same production. Since the argument strategy expression is not applied on the current tree but rather on its children, as with sequence, lifting

may be required if the argument is not a reference to a strategy attribute. Consider a production `prod` with children `a`, `b` and `c`, where `s` occurs on `a` and `c` but not `b`. When `s` is total, `all(s)` is translated as `prod(a.s, b, c.s)` to produce a total result. [4] Otherwise when `s` is partial, a partial result is computed by the translation

```
case a.s, c.s of
| just(a_s),just(c_s) ->just(prod(a_s,b,c_s))
| _, _ -> nothing()  end
```

This could also be expressed through chained monadic binds operations.

The translation of `some(s)` is similar to `all(s)`, except that `s` is allowed to fail for some of the children, which are preserved unchanged in the result. However when `s` fails for (or doesn't occur on) all children, `some(s)` will fail. Thus `some(s)` is always partial, regardless of whether `s` is total.

When `s` is total and occurs on at least one child, the translation of `some(s)` is identical to that of `all(s)`. However the translation is somewhat more complicated when `s` is partial, as the strategy must fail when all children fail, but if any succeed the strategy must succeed and include the successful results in the overall result. The translation for the above example `prod` would be

```
if a.s.isJust || c.s.isJust
then just(prod(
  fromMaybe(a, a.s), b, fromMaybe(c, c.s)))
else nothing()
```

where `isJust` is an attribute of type `Boolean` on `Maybe`.

`one` is similar to `some` except that all children to the right of the first successful child are left unchanged. When `s` is total, the translation for `one(s)` on `prod` is

```
just(prod(a.s, b, c))
```

When `s` is partial, the translation would be

```
case a.s, c.s of
| just(a_s), _ -> just(prod(a_s, b, c))
| _, just(c_s) -> just(prod(a, b, c_s))
| _, _ -> nothing()  end
```

Congruence traversals are always partial, as they only succeed for the specified production. Their translation is similar to that of `all`, except that different attributes may be accessed on different children. For example if `s1` is a partial strategy attribute occurring on the first child of `prod` and `s2` is a total strategy attribute that occurs on the third child, then the strategy expression `prod(s1, id, s2)` would be translated

```
case a.s1 of
| just(a_s1) -> just(prod(a_s1, b, c.s2))
| _ -> nothing()  end
```

The translation of **`rec`** is straightforward when **`rec`** is the outermost strategy expression, such as in

```
strategy attribute foo = rec s -> all(s) <+ ...;
```

as the recursive variable `s` can simply be replaced by `foo`, eliminating the **`rec`** expression. Lifting can be used when **`rec`** occurs elsewhere within a strategy expression.

4.3 Implementation as an Extension to Silver

Strategy attributes are implemented as a modular extension to Silver, by introducing new productions on nonterminals in the Silver specification that translate (via forwarding [39]) to existing constructs in the Silver language. This is much easier than starting from scratch and also means that all the core Silver language constructs can be used with specifications using strategy attributes; additionally, other third-party modular extensions to Silver do not need to take strategy attributes into consideration. We also reuse some Silver constructs in strategy expressions; the **`rule`** strategy uses the Silver constructs for patterns and match rules from `case` expressions. This lets us directly use the concrete syntax for patterns as seen in the for-loop-normalizing example in Section 5.3. As an extension, this work passes the modular determinism analysis [35] and the modular well-definedness analysis [17], ensuring that it will work properly with other independently developed extensions to Silver.

This does come with some trade-offs in comparison to simply implementing strategy attributes as a core feature of the Silver compiler. For example, a new **`propagate`** construct is needed that can forward to the generated aspect productions and equations implementing the strategy; simply overloading the existing `occurs on` syntax to also generate these declarations is not permitted, as the modular well-definedness analysis restricts occurrence declarations from affecting the environment. While writing explicit **`propagate`** declarations can be annoying, we believe that requiring these also satisfies a design principle of being "concise but explicit"; this pattern is seen elsewhere such as with `deriving` in Haskell. However we believe that such trade-offs are worthwhile due to the significant flexibility and ease of implementation gained as a Silver extension.

5 Applications

5.1 Evaluation of the λ-calculus

A common strategic rewriting example is evaluating the untyped λ-calculus; implementations are provided as examples by Stratego [10], Kiama [5] and reflection-based term rewriting in Silver [27] [6]. A strategy attribute implementation based on these is shown in Appendix A.1. It is as expected but uses attributes to calculate free variables, used in conditional guards on some rules.

[4] Any children on which the argument does not occur are copied unchanged; they are "invisible" for the purposes of the traversal operation.

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

[6] Available at http://melt.cs.umn.edu/ and https://github.com/melt-umn/lambda-calculus, archived at https://doi.org/10.13020/xcfv-5k29.

We have chosen to compare the performance of strategy attributes with Kiama and reflection-based term rewriting feature of Silver since these systems provide a limited integration of attributes and strategies (but lacking support for inherited attributes in rules.) Three slight variations of λ-calculus were implemented using these systems. Strategy attributes provided a 2 to 3× performance improvement over Kiama, with one outlier at 13×, and an approximately 8.5× improvement over term rewriting in Silver. Note that part of the success of Kiama is that it is a library designed to be simple and easy to understand. Both systems use reflection in building new trees, and the rewriting process is more dynamic in that is supports strategies created at runtime. Dynamic strategies are not supported by strategy attributes, which are compiled to higher-order attributes; these differences likely account for much of the performance gain.

5.2 Regular Expression Matching via Derivatives

Regex matching can be implemented in a functional setting through an approach known as Brzozowski derivatives [7]. Here the derivative of a regular expression r with respect to some character c is defined as the regular expression r' such that r' matches s for every string cs matched by r. If one repeatedly takes the derivative of r with respect to each character in a string s, then r matches s if and only if the resulting regex is nullable (*i.e.* matches the empty string.)

This can be elegantly implemented in attribute grammars with a higher-order synthesized attribute on the abstract syntax of regular expressions to compute the derivative with respect to some character provided as an inherited attribute. A strategy attribute can be used to improve performance by simplifying the regex after each derivative with identities such as $s\epsilon = s$ or $\emptyset* = \emptyset$. The use of strategy attributes provided a 15× performance speedup over a similar implementation using Silver's reflection-based term rewriting mechanism. The complete implementation using strategy attributes is provided in Appendix A.2.

5.3 Normalizing for-loops for Pattern Matching

One example use of strategy attributes may be found in the implementation of the ableC-Halide extension [19] [7] to ableC [18], inspired by the Halide [34] C++ embedded DSL. This extension allows for iterative computations consisting of multiple nested loops to be expressed separately from optimizing transformations (such as unrolling, tiling or parallelizing), allowing for more readable code and greater ease of experimentation with various transformations without fear of introducing errors.

An example use of the extension to perform an optimized matrix multiplication can be found in Figure 12. Here the computation to be performed is specified using ordinary C

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

```
1  transform {
2    for (int i = 0; i < m; i++)
3      for (int j = 0; j < n; j++) {
4        c[i][j] = 0;
5        for (int k = 0; k < p; k++)
6          c[i][j] += a[i][k] * b[k][j];
7      }
8  } by {
9    split i into (int i_outer,
10     int i_inner : (m - 1) / N_THREADS + 1);
11   parallelize i_outer into
12     N_THREADS threads;
13   tile i_inner, j into (TILE_DIM, TILE_DIM);
14 }
```

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

statements (lines 2—7), while a series of optimizing transformations on contained `for`-loops (each identified by their loop variable) are written using a custom DSL (lines 9—14.) For example, splitting a loop converts it into multiple nested loops where all but the outermost run for a constant number of iterations, while unrolling a loop requires duplicating its body a number of times in sequence. Tiling, parallelizing, and vectorizing transformations have the expected behavior.

Computing the translation of a **`transform`** statement requires `for`-loops to be recognized; this can be done by recursively pattern matching on the abstract syntax. However before this can be done, all loops must be normalized to the form `for (`*type* `i = 0; i <` *limit*`; i++)` *body*; for some variable `i`. For example the loop

```
for (int i = 2; -5 <= i; i -= 2) x += i;
```

should be normalized to

```
for (int i = 0; i < 4; i++) {
  int old_i = 2 - i * 2; x += old_i; }
```

This can be achieved using strategy attributes. One rule, of many, used in this process is shown in Figure 13; here uses of the <= operator in loop conditions are replaced with <.

In this example, rewrite rules are defined over the abstract syntax of ableC. Concrete syntax patterns and expressions [26, 27] allow for messy syntax trees to be specified using concrete syntax, permitting significant improvements in code size, readability and maintainability. These concrete patterns and expressions are extensions to Silver that forward to the equivalent (verbose) versions that use abstract syntax; due to the modular nature of extensions built with forwarding [39], the implementation of strategy attributes does not require any special handling for concrete patterns.

Discussion: Another advantage of tree rewriting over undecorated term rewriting is in regard to composable language

```
1  partial strategy attribute preprocessLoop = rule on Stmt of  -- Normalize loop conditions
2   | ableC_Stmt{ for ($Decl{init} $Name{i} <= $Expr{limit}; $Expr{iter}) $Stmt{b} } ->
3     ableC_Stmt{ for ($Decl{init} $Name{i} < $Expr{limit} + 1; $Expr{iter}) $Stmt{b} }...end;
```

Figure 13. One of the many rewrite rules used to normalize loops in the ableC-Halide extension.

extensions. Translating strategies into attributes means that interaction with other features of attribute grammars, such as forwarding [39] or a modular well-definedness analysis [17], is already handled properly. For example, consider an independent language extension providing a new `forall` statement for concisely specifying nested loops. This would let one express lines 2—3 of Figure 12 more concisely as `forall (int i : m, int j : n)`, which might forward to

```
for (int i = 0; i<m; i+=1) for (int j = 0; j<n; j+=1)
```

In order for these loops to be transformed by the ableC-Halide extension, they must first be normalized by rewriting `+= 1` to `++`. Since the strategy attributes for normalization (*e.g.* the one in Figure 13) are defined on the `for`-statement host language production to which the `forall`-statement extension production forwards, normalization will automatically happen on the forward tree as desired. Conversely an implementation [27] of the same rewrite rules in Silver's undecorated term rewriting system fails to recognize and properly transform this `forall` extension production.

5.4 Optimizing Strategy Translation

Another interesting use of strategy attributes is internally within their own implementation, to optimize strategy expressions on a per-production basis before translation. For example in Figure 9, the equations in production `const` for `optimizeStep` and `optimize` have been reduced to `nothing()` and `top` (line 13) since `optimizeStep` will never succeed for this production, and thus `optimize` will have no effect.

The rules for optimizing strategy expressions are divided into two categories: generic rules that are applicable regardless of the context, and production-dependent optimizations. Generic rules correspond to basic identities such as

$$\texttt{fail} \texttt{<*}\ s \longrightarrow \texttt{fail} \quad (11) \qquad s_1 \texttt{<+}\ s_2 \longrightarrow s_1 \ \text{ if } s_1\ \textit{total} \quad (14)$$

$$\texttt{id} \texttt{<*}\ s \longrightarrow s \quad (12) \qquad \texttt{all(fail)} \longrightarrow \texttt{fail} \quad (15)$$

$$\texttt{fail} \texttt{<+}\ s \longrightarrow s \quad (13) \qquad \texttt{rec}\ n\ \texttt{->}\ s \longrightarrow s \ \text{ if } n \notin \mathit{fv}(s) \quad (16)$$

These are implemented as a single rule strategy expression on the `StrategyExpr` nonterminal (as with rules 1—7 in Figure 3), and (except for rule 16, which computes the free variables in s) do not involve the use of attributes.

Other optimizations, such as eliminating non-matching rules and congruences, do depend on the current production and use an inherited attribute `frame` containing the production's name and signature. However it is not safe to apply these optimizations everywhere in a strategy expression, as some sub-expressions (such as arguments to traversals or the right operand of `<*`) are not computed on the current tree. This can be avoided when optimizing a strategy expression by using congruences to apply generic rules everywhere and production-specific ones only where it is safe to do so. The full implementation is shown in Figure 17 of Appendix A.4.

These optimizations provided a roughly 15 to 20 percent speedup, respectively for the regex and λ-calculus examples, in comparison to the unoptimized translation. While significant, this is less than might be expected, largely due to the just-in-time compilation approach of the Java Virtual Machine (Silver is translated into Java.) This allows for reduced overhead of branches that are consistently not taken, such as those arising from a case pattern that always fails to match. Thus if left undone at compile time, many of the optimizations we present will still effectively happen at run time with only a small increase in overhead.

6 Related Work

As mentioned in Section 1, both attribute grammars and (strategic) term rewriting systems have a long history. Although these formalisms have largely been considered separately there has been some work to integrate them.

The JastAdd [11] AG system supports rewriting and rewrite-rules that can reference attributes on trees as is done here with strategy attributes. When a tree is rewritten, its attributes may be recomputed based on its context. However, strategies are not supported. All rewrite-rules in the specification can be applied, if their guard holds. JastAdd is object-oriented in that it allows deeper class hierarchies than in traditional grammars and these can be used to restrict the application of rewrite rules to certain sub-classes and to prioritize rules based on the sub-class relationship of the rules' right hand sides. JastAdd also supports circular attributes [12]; these attributes, via circular equations, depend on their own value in a previous step in a fixed-point computation. JastAdd introduced reference attributes and was the first to integrate circular and reference attributes [29], and then integrate its approach to rewriting [38]. Silver has reference attributes and these work seamlessly with strategy attributes, but it does not have circular attributes and we have not considered integrating these with strategy attributes. Since strategies are translated into traditional higher-order attributes, the integration may be straightforward, but it may raise questions about how they are effectively used together.

Kiama [36] is an embedded DSL in Scala for language processing tasks. It was the first to support both attribute grammars and strategic term rewriting (in the same style of Stratego) and was partly the inspiration of this work.

In Kiama the tree structure and its attribution are separate, but linked, data structures [37]. Rewriting takes place on the tree structures that can access the attribution during rewriting, but new intermediate trees constructed during rewriting are not considered as part of the whole tree until the rewriting completes, and thus are not given contextual information which corresponds to the inherited attributes in our approach. The attribution and rewriting processes are intentionally seen as more-or-less separate ones and provide an integration of the two is less complete as with strategy attributes. For this reason, tasks like the expression inlining in Section 3.3 that use attribute values on new intermediate trees created during rewriting may be less directly specified.

Aster [23] is a system that uses strategic term-rewriting as a basis for implementing various schemes for propagating attribute information in trees, such as reference and remote attributes, chain attributes for threading information left-to-right through a tree, and several others. Although based on strategic rewriting, the exploration of attribute grammars with rewriting is left as future work.

Silver has a reflection library for converting well-sorted trees into a generic representation [26] and a strategic *term* rewriting system [27] is built on top of that. But it does not support the use of decoration of new trees with inherited attributes as strategy attributes do and is thus limited in its application. There is an implementation of λ-calculus evaluation in the same style as discussed in Section 5.1 [27]. Rewrite rules can access synthesized attributes that do not depend on any inherited attributes by reifying the well-sorted tree from its generic representation. This is rather inefficient as the strategy attribute implementation of the λ-calculus evaluator was on average $8.5\times$ faster than this reflection based one.

It is sometimes helpful to construct new strategies at runtime. This is used in Stratego [31], for example, to create new rewrite rules based on program variables to propagate contextual information for concerns such as lexical scope. Kiama and reflection-based rewriting in Silver also support dynamic strategies, but the compile-time translation of strategy attributes to higher-order attribute precludes them. However, we would contend that inherited attributes often provide a more direct means for specifying contextual information for issues such as scoping, name-binding, and type-checking. In fact, it is not clear that dynamic rewrite rules are the preferred mechanism for contextual information in Stratego. It is now part of the Spoofax [22] language workbench which includes other domain-specific languages, such as NaBL [25] and it successors for such purposes.

7 Discussion and Conclusion

Tree rewriting with strategy attributes does have some limitations in comparison to traditional rewriting on undecorated terms. Since trees are required to be well-sorted, translations between different sorts are not possible without some inelegant adapter productions; however higher-order attributes often provide a more elegant alternative to rewriting for this type of problem. Many term rewriting systems provide a notion of *associative* and *commutative* rewriting over list terms. This is not possible in our implementation of strategy attributes, as lists in Silver are a parametric type that cannot be decorated with attributes and type-specific collection nonterminals such as `Exprs` are not recognized as list-like structures. Additionally, utility strategies here are language extensions, not library functions as in Stratego and Kiama that are more easily written by users.

Section 5.3 showed how tree rewriting takes forwarding into consideration, as strategy attributes are evaluated by default on the forwarded-to (translated) tree. But some transformations, *e.g.* instantiating C++-style template extensions [19], should be done on the original (forwarding) construct. This is not possible with tree rewriting, as strategy attributes would need to be propagated on unknown third-party extension productions, whereas term rewriting would treat all productions uniformly by default.

Another future area of development is in detecting strategies that inefficiently repeat work. Some such strategies (*e.g.* the `optimizeInline` strategy as defined in Section 3.3) would inefficiently repeat traversals even in systems such as Stratego that work on undecorated terms. However the integration of attributes creates another class of potential inefficiencies when rewrite rules demand attribute values, which for some strategies can cause the children to be re-decorated with attributes an exponential number of times. This latter class of performance issues bears some resemblance to performance problems that sometimes arise with the use of forwarding, so a solution to automatically finding cases of re-decoration would be broadly applicable.

Despite these limitations, strategy attributes provide, in our view, a compelling integration of strategic rewriting with attribute grammars. This approach reduces boilerplate in attribute grammar specifications by allowing complex transformations to be concisely specified in the language of strategies, while permitting easy access to contextual information in rewrite rules using inherited attributes. We have demonstrated that competitive performance can be attained by static compilation and optimization. Finally we have presented several applications of strategy attributes, illustrating the use of strategy attributes in creating non-trivial yet compact and readable attribute grammar specifications.

Acknowledgments

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

A Additional Details on Applications

In this appendix we present additional details on the implementations of the applications discussed in Section 5. The full source code of these can be found on the MELT website (melt.cs.umn.edu) and in the artifact accompanying this paper.

A.1 Evaluation of the λ-calculus

An implementation of the untyped λ-calculus based on strategy attributes is shown in Figure 14, based on Stratego [10] and Kiama [3] implementations of the same. Here substitution is implemented by means of an additional production for representing intermediate *let*-terms (line 16, named `letT` to avoid conflicting with Silver's own `let` keyword). This term is only introduced by β-reduction (line 21) and is subsequently distributed through the term using the `letDist` strategy (line 25). `letT` only exists during normalization. On line 37, `letT` bindings that are not referenced in their body are discarded. This is checked with the use of a synthesized attribute `freeVars` on `Term`.

The total strategy attribute `evalInnermost` (line 41) performs a fully eager evaluation of a term, including normalizing under lambda expressions, which is typically not desirable or guaranteed to terminate - thus more precise control over the traversal process is desired. This can be achieved by using congruences. For example, a term is in weak head normal form if all applications that do not occur within an abstraction body have been reduced; the `evalWHNF` strategy on line 44 of Figure 14 implements this by only recursing into the sub-terms of applications and lets, but does not perform β-reduction within the bodies of abstractions.

A.2 Regular Expression Matching via Derivatives

The abstract syntax of regular expressions is represented by the `Regex` nonterminal in Figure 15, with productions for $\emptyset$ (the regex that does not match anything), ϵ (matching the empty string), single characters, sequence, alternative and the Kleene $*$ operation.

The Brzozowski derivative of a `Regex` is implemented by the `deriv` synthesized attribute (line 2), with respect to some character specified by the `wrt` autocopy inherited attribute[8] (line 3.) For example, the derivative of both `empty` (line 7) and `epsilon` (line 10) is `empty()`, since neither of these match any string containing a single character. The derivative of `char(c)` with respect to `c` is `epsilon()`, since removing `c` from a string matched by this regex must leave only the empty string, but is `empty()` with respect to any other character (line 13.) The derivative of `alt(r1, r2)` corresponds to the

[8] Autocopy inherited attributes are just like regular inherited attributes, except that their values are automatically copied down the tree when explicit equations are not specified. This allows for a significant amount of boilerplate code to be avoided.

```
1  synthesized attribute freeVars::[String];
2  nonterminal Term with freeVars,
3    beta, letDist, evalInnermost, evalWHNF;
4  abstract production var
5  top::Term ::= id::String
6  { top.freeVars = [id];
7  }
8  abstract production abs
9  top::Term ::= id::String body::Term
10 { top.freeVars = remove(id,
11                         body.freeVars); }
12 abstract production app
13 top::Term ::= t1::Term t2::Term
14 { top.freeVars = t1.freeVars ++
15                  t2.freeVars;  }
16 abstract production letT
17 top::Term ::= id::String t::Term
18               body::Term
19 { top.freeVars = t.freeVars ++
20              remove(id, body.freeVars); }
21 partial strategy attribute beta =
22   rule on Term of
23   | app(abs(x, e1), e2) -> letT(x, e2, e1)
24   end;
25 partial strategy attribute letDist =
26   rule on Term of
27   | letT(x, e, var(y)) when x == y -> e
28   | letT(x, e, var(y)) -> var(y)
29   | letT(x, e0, app(e1, e2)) ->
30     app(letT(x, e0, e1), letT(x, e0, e2))
31   | letT(x, e1, abs(y, e2)) ->
32     let z::String = freshVar() in
33       abs(z, letT(x, e1,
34                   letT(y, var(z), e2)))
35     end
36   | letT(x, _, e)
37     when !contains(x, e.freeVars) -> e
38   end;
39 propagate beta, letDist on Term;
40
41 strategy attribute evalInnermost =
42   innermost(beta <+ letDist);
43
44 strategy attribute evalWHNF =
45   try(app(evalWHNF, evalWHNF) <+
46       letT(id, evalWHNF, evalWHNF)) <*
47   try((beta <+ letDist) <* evalWHNF);
48
49 propagate evalInnermost, evalWHNF on Term;
```

Figure 14. An implementation of the λ-calculus based on strategy attributes.

```
1  synthesized attribute nullable::Boolean;
2  synthesized attribute deriv::Regex;
3  autocopy attribute wrt::Integer; -- Encodes a UTF-16 character value
4  nonterminal Regex with nullable, deriv, wrt, simpl, simplDeriv;
5  abstract production empty
6  top::Regex ::=
7  { top.nullable = false; top.deriv = empty(); }
8  abstract production epsilon
9  top::Regex ::=
10 { top.nullable = true; top.deriv = empty(); }
11 abstract production char
12 top::Regex ::= c::Integer
13 { top.nullable = false; top.deriv = if c == top.wrt then epsilon() else empty(); }
14 abstract production seq
15 top::Regex ::= r1::Regex r2::Regex
16 { top.nullable = r1.nullable && r2.nullable;
17   top.deriv = alt(seq(r1.deriv, r2), if r1.nullable then r2.deriv else empty()); }
18 abstract production alt
19 top::Regex ::= r1::Regex r2::Regex
20 { top.nullable = r1.nullable || r2.nullable; top.deriv = alt(r1.deriv, r2.deriv); }
21 abstract production star
22 top::Regex ::= r::Regex
23 { top.nullable = true; top.deriv = seq(r.deriv, top); }
24
25 strategy attribute simpl = innermost(
26   rule on Regex of
27   | seq(empty(), r) -> empty()              | seq(r, empty()) -> empty()
28   | seq(epsilon(), r) -> r                  | seq(r, epsilon()) -> r
29   | alt(empty(), r) -> r                    | alt(r, empty()) -> r
30   | alt(epsilon(), r) when r.nullable -> r  | alt(r, epsilon()) when r.nullable -> r
31   | star(empty()) -> epsilon()              | star(epsilon()) -> epsilon()
32   end);
33 strategy attribute simplDeriv = deriv <* simpl;
34 propagate simpl, simplDeriv on Regex;
35
36 function matchStep
37 Regex ::= r::Regex c::Integer
38 { r.wrt = c; return c.simplDeriv; }
39 function matchesRegex
40 Boolean ::= r::Regex s::String
41 { return foldl(matchStep, stringToChars(s)).nullable; }
```

Figure 15. An implementation of regex matching using Brzozowski derivatives.

alternative of the derivatives (line 20), and the derivative of `star(r)` is the derivative of `r` followed by `star(r)` (line 23.)

A separate synthesized attribute `nullable` (line 1) determines whether the `Regex` matches the empty string; for example `epsilon` and `star` are always nullable, while `seq(r1, r2)` is nullable when both `r1` and `r2` are nullable, and `alt(r1, r2)` is nullable when either is nullable. This attribute is used in computing the derivative of `seq(r1, r2)`: removing a character from the front of this `Regex` can either correspond to removing it from `r1` or, if `r1` is nullable, removing it from `r2` (line 17.)

As mentioned previously, one can check if a string matches a regex by iteratively computing the derivative with respect to each character in the string, and checking whether the resulting regex is nullable. However, a naive implementation would result in the regex increasing in size after every derivative due to the accumulation of `empty()` terms, leading to linear space and quadratic time complexity in the length of

the string, or worse. One solution is an approach known as *smart constructors* [32], in which direct calls to the constructors are replaced by functions that pattern match on their arguments and perform the needed simplifications.

Strategy attributes provide another (arguably more elegant) solution in the context of attribute grammars. Here the `simpl` strategy attribute (lines 25—32) performs an innermost traversal of the `Regex`, simplifying it according to a number of basic identities. `simpl` is then used to define the `simplDeriv` strategy attribute, that computes the derivative of a `Regex` and simplifies the result. Note that `deriv` is not a strategy attribute, however it may still be used within a strategy expression because it has the same type as a total strategy attribute for the `Regex` nonterminal on which it occurs.

The functions `matchesRegex` and `matchStep` (lines 36—41) drive the actual string matching process; here a fold operation is used, incrementally decorating `r` with each character as the value for the `wrt` attribute, and computing `simplDeriv` on the resulting tree.

A.3 Normalizing `for`-loops for Pattern Matching

The `for`-loop normalization rewrite rules and strategies presented here are a strategy attribute re-implementation of the same rules and strategies, originally implemented [27] using Silver's reflection-based library and language extension for rewriting on undecorated terms. Both versions can be found in the ableC-Halide extension [9].

This rewriting process is divided into two passes. First, an initial downward pass fully expands the condition and update expressions in loops to only use the operators `<`, `>`, `+=`, and `-=` at the top level; for example `a <= b` is rewritten to `a < b + 1`. This is done by a strategy attribute named `preprocessLoop` (lines 21—25.)

A second, bottom-up pass performs the final normalization; this is implemented by the `transLoop` strategy attribute (lines 26—47.) In this pass, loops that range from 0 and step by 1 can be directly translated to use the `++` increment operator, and do not require any additional processing (line 28.) However loops that have a different initial or step value require a more complex transformation; the loop variable is changed to start at 0 and increment by 1 (line 40), and a new variable is introduced to compute the value of the original loop variable from the rewritten loop variable (line 42.)[10] All occurrences of the original loop variable in the body are then replaced by the new loop variable (line 44.)

This is done by a separate strategy attribute `renamed` in conjunction with autocopy inherited attributes `target` and `replacement` (lines 1—8.) `renamed` makes use of the `allTopDown` strategy, which traverses down the tree until the its argument first succeeds (in this case when the tree is a name that matches `target`.) The original tree is then reconstructed with name replaced by the new name provided via `replacement`.

At every point during the upward and downward passes, we wish to simplify loop expressions involving constants as much as possible, maximizing the variety of loops that can be successfully normalized. The `simplifyExpr` total strategy attribute (line 15) simplifies all such expressions within a tree using the `simplifyExprStep` partial strategy attribute (lines 10—14.) Since we only wish to simplify expressions within the definition of the loop and not in the body, the `simplifyLoopExpr` strategy attribute (line 19) is used in order to achieve this, by means of a congruence traversal over the `forDeclStmt` production.

The overall rewriting process is driven by the total strategy attribute `normalizeLoop` (line 48). This uses the `downUp` strategy to apply the `preprocessLoop` and `transLoop` strategies in top-down and bottom-up passes, performing loop expression simplification before every preprocessing step and after each transformation step.

A.4 Optimizing Strategy Translation

More of the rules and strategies used in optimizing strategy expressions are shown in Figure 17. The rules for optimizing strategy expressions are divided into two categories: generic rules for algebraic simplification that are applicable regardless of the context, and production-dependent optimizations. Generic optimizations are implemented by the `genericStep` strategy attribute; not shown here are rules for optimizing references to names, such as inlining references to strategy attributes.

Other optimizations, implemented by the `prodStep` strategy attribute, are production-dependent; for example, eliminating rules and congruences that do not match the current production. These optimizations depend on the synthesized attribute `matchesFrame`, true only if the rule's pattern is able to match, which in turn depends on an inherited attribute `frame` containing the current production's name and signature.

Rule strategy expressions are represented by the production `rewriteRule`; this has the signature

```
StrategyExpr ::= Name TypeExpr MatchRuleList
```

The `prodStep` strategy replaces an entire **`rule`** strategy expression with `fail` if all patterns in the match rule list fail to match, or alternatively deletes a single non-matching match rule from the list. This is done by using a congruence on the `rewriteRule` production to traverse the match rule list with the `onceBottomUp(s)` strategy, defined as

```
rec x -> one(x) <+ s.
```

Additional rules deal with traversal strategies where the argument does not occur on any of the children; in this case `all` is replaced with `id` and `some`/`one` becomes `fail`.

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

[10] Loops that work in reverse order by decrementing the loop variable are handled similarly by a separate case in the `transLoop` rule, not shown.

```
autocopy attribute target::String;
autocopy attribute replacement::String;
strategy attribute renamed = allTopDown(
  rule on top::Name of
  | name(n) when n == top.target -> name(top.replacement, location=top.location)
  end);
attribute target, replacement, renamed occurs on Decl, Stmt, Expr, Name, ...;
propagate renamed on Decl, Stmt, Expr, Name, ...;

partial strategy attribute simplifyExprStep =
  rule on Expr of   -- Simplify expressions as much as possible
  | ableC_Expr { $Expr{intExpr(a)} + $Expr{intExpr(b)} } -> intExpr(a + b)
  | ableC_Expr { $Expr{intExpr(a)} / $Expr{intExpr(b)} } when b != 0 -> intExpr(a / b)
  | ... end;
strategy attribute simplifyExpr = innermost(simplifyExprStep);
attribute simplifyExprStep, simplifyExpr occurs on Expr;
propagate simplifyExprStep, simplifyExpr        on Expr;

partial strategy attribute simplifyLoopExpr =
  forDeclStmt(simplifyExpr, simplifyExpr, simplifyExpr, id);
partial strategy attribute preprocessLoop =
  rule on Stmt of   -- Normalize condition operators
  | ableC_Stmt { for ($Decl{init} $Name{i} <= $Expr{limit}; $Expr{iter}) $Stmt{b} } ->
    ableC_Stmt { for ($Decl{init} $Name{i} < $Expr{limit} + 1; $Expr{iter}) $Stmt{b} }
  | ... end;
partial strategy attribute transLoop =
  rule on top::Stmt of
  | ableC_Stmt {   -- Restore increment operator on loops that are otherwise-normal
      for ($TypeExpr{t} $Name{i1} = 0; $Name{i2} < $Expr{n}; $Name{i3} += 1) $Stmt{b}
    } when i1.name == i2.name && i1.name == i3.name   ->
    ableC_Stmt {
      for ($TypeExpr{t} $Name{i1} = 0; $Name{i2} < $Expr{n}; $Name{i3}++) $Stmt{b}
    }
  | ableC_Stmt {   -- Normalize loops with nonstandard initial or step values
      for ($TypeExpr{t} $Name{i1} = $Expr{initial};
           $Name{i2} < $Expr{limit}; $Name{i3} += $Expr{step}) $Stmt{b}
    } when i1.name == i2.name && i1.name == i3.name   ->
      let newName::String = freshVarName()
      in ableC_Stmt {
        for ($TypeExpr{t} $Name{i1} = 0;
             $Name{i2} < ($Expr{limit} - $Expr{initial}) / $Expr{step}; $Name{i3}++) {
          typeof($Name{i1}) $name{newName} = $Expr{initial} + $Name{i1} * $Expr{step};
          $Stmt{decorate b with {target = i1.name; replacement = newName;
                                 env = top.env;}.renamed}    }
      } end
  | ... -- Similar rules as above for loops that count downward
  end;
strategy attribute normalizeLoop =
  downUp(try(simplifyLoopExprs <* repeat(preprocessLoop)),
         try(transLoop <* simplifyLoopExprs));
attribute simplifyLoopExprs, preprocessLoop, transLoop, normalizeLoop occurs on Stmt;
propagate simplifyLoopExprs, preprocessLoop, transLoop, normalizeLoop        on Stmt;
```

Figure 16. More of the rewrite rules used to normalize loops in the ableC-Halide extension.

```
-- Production-independent optimizations
partial strategy attribute genericStep =
  rule on StrategyExpr of
  | sequence(fail(), _) -> fail()
  | sequence(_, fail()) -> fail()
  | sequence(id(), s) -> s
  | sequence(s, id()) -> s
  | choice(fail(), s) -> s
  | choice(s, fail()) -> s
  | choice(s, _) when s.isTotal -> s
  | allTraversal(id()) -> id()
  | someTraversal(fail()) -> fail()
  | oneTraversal(fail()) -> fail()
  | congruenceTraversal(_, ss) when ss.containsFail -> fail()
  | recComb(n, s) when !contains(n.name, s.freeVars) -> s
  end;
-- Production-dependent optimizations
partial strategy attribute prodStep =
  rule on top::StrategyExpr of
  | rewriteRule(_, _, ml) when !ml.matchesFrame -> fail()
  | congruenceTraversal(p, _) when p.fullName != top.frame.fullName -> fail()
  | allTraversal(s)  when !matchesChild(s, top.frame) -> id()
  | someTraversal(s) when !matchesChild(s, top.frame) -> fail()
  | oneTraversal(s)  when !matchesChild(s, top.frame) -> fail()
  end <+
  rewriteRule(id, id, onceBottomUp(
      rule on MatchRuleList of
      | mRuleList_cons(h, _, t) when !h.matchesFrame -> t
      end));
strategy attribute simplify = innermost(genericStep);
strategy attribute optimize =
  (sequence(optimize, simplify) <+ choice(optimize, optimize) <+
   allTraversal(simplify) <+ someTraversal(simplify) <+ oneTraversal(simplify) <+
   congruenceTraversal(simplify) <+ recComb(id, optimize) <+ id) <*
  try((genericStep <+ prodStep) <* optimize);
```

Figure 17. Some of the rewrite rules used to optimize strategy expressions.

The overall optimization process is driven by the `optimize` strategy attribute. Note that production-specific optimizations are not safe to apply arbitrarily in a strategy expression, as some strategy combinators apply their argument to a different tree (as with congruences and the right operand to sequence.) Thus the strategy may ultimately be applied on a different production and so it is unsafe to apply the `prodStep` optimization. However it *is* always safe to apply the production-independent optimizations specified by `genericStep`; the strategy attribute `simplify` performs these optimizations in an innermost pass through the whole tree. `optimize` makes this distinction by explicitly traversing the `StrategyExpr` tree using congruences rather than `innermost`; the `optimize` strategy is applied recursively to child strategy expressions known to be applied to the same production, while `simplify` is applied to all other children.

References

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

[2] Arthur Baars, Doaitse Swierstra, and Andres Loh. UU AG system reference manual. 2004. URL http://www.cs.uu.nl/~arthurb/data/AG/AGman.html.

[3] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on Java. In Franz Baader, editor, *Term Rewriting and Applications*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2007. doi: 10.1007/978-3-540-73449-9_5.

[4] Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. Elan from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155 – 185, 2002. doi: 10.1016/S0304-

3975(01)00358-9. Rewriting Logic and its Applications.

[5] John Tang Boyland. Remote attribute grammars. *Journal of the ACM*, 52(4):627–687, 2005. doi: 10.1145/1082036.1082042.

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

[7] Janusz A Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[8] James R. Cordy. TXL - a language for programming language tools and applications. *Electronic Notes in Theoretical Computer Science*, 110:3 – 31, 2004. doi: 10.1016/j.entcs.2004.11.006. Proceedings of the Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA 2004).

[9] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[10] Eelco Dolstra and Eelco Visser. Building interpreters with rewriting strategies. *Electronic Notes in Theoretical Computer Science*, 65(3):57–76, 2002. doi: 10.1016/S1571-0661(04)80427-4.

[11] Torbjörn Ekman and Görel Hedin. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69:14–26, December 2007. doi: 10.1016/j.scico.2007.02.003.

[12] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. *ACM SIGPLAN Notices*, 21(7), 1986. doi: 10.1145/13310.13320.

[13] Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane, and William M. Waite. Eli: a complete, flexible compiler construction system. *CACM*, 35(2):121–130, 1992. doi: 10.1145/129630.129637.

[14] Görel Hedin. Reference attribute grammars. *Informatica*, 24(3):301–317, 2000.

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

[16] M. Jourdan, D. Parigot, Julié, O. Durin, and C. Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *Conference on Programming Languages Design and Implementation*, pages 209–222, 1990. doi: 10.1145/93548.93568. Published as *ACM SIGPLAN Notices*, 25(6).

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

[18] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. Reliable and automatic composition of language extensions to C: The ableC extensible language framework. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):98:1–98:29, October 2017. doi: 10.1145/3138224.

[19] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. Reliable and automatic composition of language extensions to C — supplemental material. Technical Report 17-009, University of Minnesota, Department of Computer Science and Engineering, 2017. Available at https://www.cs.umn.edu/research/technical_reports/view/17-009.

[20] U. Kastens, B. Hutt, and E. Zimmermann. *GAG: A Practical Compiler Generator*, volume 141 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982. doi: 10.1007/BFb0034297.

[21] Uwe Kastens. Ordered attributed grammars. *Acta Informatica*, 13:229–256, 1980. doi: 10.1007/BF00288644.

[22] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)*, OOPSLA. Association of Computing Machinery, 2010. doi: 10.1145/1869459.1869497.

[23] Lennart C. L. Kats, Anthony M. Sloane, and Eelco Visser. Decorated attribute grammars: Attribute evaluation meets strategic programming. In Oege de Moor and Michael I. Schwartzbach, editors, *Compiler Construction*, volume 5501 of *Lecture Notes in Computer Science*, pages 142–157. Springer, 2009. doi: 10.1007/978-3-642-00722-4_11.

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

[25] Gabriël D.P. Konat, Vlad A. Vergu, Lennart C.L. Kats, Guido H. Wachsmuth, and Eelco Visser. The spoofax name binding language. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 79–80, New York, NY, USA, 2012. Association of Computing Machinery. doi: 10.1145/2384716.2384748.

[26] Lucas Kramer, Ted Kaminski, and Eric Van Wyk. Reflection in attribute grammars. In *Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts & Experience (GPCE)*, pages 48–60. ACM, 2019. doi: 10.1145/3357765.3359517.

[27] Lucas Kramer, Ted Kaminski, and Eric Van Wyk. Reflection of terms in attribute grammars: Design and applications. *Journal of Computer Languages*, 2020. To appear.

[28] M. Kuiper and Saraiva J. LRC — a generator for incremental language-oriented tools. In *7th International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 298–301. Springer-Verlag, 1998. doi: 10.1007/BFb0026440.

[29] Eva Magnusson and Görel Hedin. Circular reference attributed grammars - their evaluation and applications. *Science of Computer Programming*, 68(1):21–37, 2007. doi: 10.1016/j.scico.2005.06.005. Special Issue on the ETAPS 2003 Workshop on Language Descriptions, Tools and Applications (LDTA '03).

[30] Marjan Mernik, Nikolaj Korbar, and Viljem Žumer. Lisa: a tool for automatic language implementation. *SIGPLAN Not.*, 30(4):71–79, 1995. doi: 10.1145/202176.202185.

[31] Karina Olmos and Eelco Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In *Proc. 14th Intl. Conf. on Compiler Construction*, volume 3443 of *Lecture Notes in Computer Science*, pages 204–220. Springer, 2005.

[32] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *Journal of Functional Programming*, 19(2):173–190, 2009. doi: 10.1017/S0956796808007090.

[33] Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995. doi: 10.1145/210376.197409.

[34] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 519–530, New York, NY, USA, 2013. ACM. doi: 10.1145/2491956.2462176.

[35] August Schwerdfeger and Eric Van Wyk. Verifiable composition of deterministic grammars. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 199–210, New York, NY, USA, June 2009. ACM. doi: 10.1145/1542476.1542499.

[36] Anthony M. Sloane. Lightweight language processing in Kiama. In *Proceedings of the 3rd summer school on Generative and Transformational Techniques in Software Engineering III (GTTSE '09)*, volume 6491

of *Lecture Notes in Computer Science*, pages 408–425. Springer, 2011. doi: 10.1007/978-3-642-18023-1_12.

[37] Anthony M. Sloane, Matthew Roberts, and Leonard G. C. Hamey. Respect your parents: How attribution and rewriting can get along. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 191–210. Springer, 2014. doi: 10.1007/978-3-319-11245-9_11.

[38] Emma Söderberg and Görel Hedin. Declarative rewriting through circular nonterminal attributes. *Computer Languages, Systems & Structures*, 44:3 – 23, 2015. doi: 10.1016/j.cl.2015.08.008. Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[39] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proceedings of the 11th Conference on Compiler Construction (CC)*, volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer-Verlag, 2002. doi: 10.1007/3-540-45937-5_11.

[40] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39–54, January 2010. doi: 10.1016/j.scico.2009.07.004.

[41] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001. doi: 10.1007/3-540-45127-7_27.

[42] Harold H. Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 131–145. ACM, 1989. doi: 10.1145/73141.74830.