Fusing Logic And Control With Local Transformations: An Example Optimization

By: Patricia Johann and Eelco Visser

Abstract:
Programming supports the separation of logical concerns from issues of control in program construction. While this separation of concerns leads to reduced code size and increased reusability of code, its main disadvantage is the computational overhead it incurs. Fusion techniques can be used to combine the reusability of abstract programs with the efficiency of specialized programs. In this paper we illustrate some of the ways in which rewriting strategies can be used to separate the definition of program transformation rules from the strategies under which they are applied. Doing so supports the generic definition of program transformation components. Fusion techniques for strategies can then be used to specialize such generic components. We show how the generic innermost rewriting strategy can be optimized by fusing it with the rules to which it is applied. Both the optimization and the programs to which the optimization applies are specified in the strategy language Stratego. The optimization is based on small transformation rules that are applied locally under the control of strategies, using special knowledge about the contexts in which the rules are applied.

1 Introduction

Abstract programming techniques support the generic definition of algorithmic functionality in such a way that different configurations of algorithms can be obtained by plugging together generic components. As a result, these components can be reused in many instances and in many different combinations. The advantages of abstract programming are reduced code size and increased
reusability of programs. A disadvantage is that the separation of concerns it supports can introduce considerable computational overhead. In contrast, code for specific problem instances can effectively intermingle logic and control to arrive at more efficient implementations than are possible generically. The challenge of abstract programming is to maintain a high-level separation of concerns while simultaneously achieving the efficiency of intermingled programs.

Fusion techniques mitigate the tension between modularity and efficiency by automatically deriving intermingled efficient versions of programs from their abstract composite versions. For example, in deforestation of functional programs, intermediate data structures are eliminated by fusing together function compositions \[6,11,17\]. Fusion also enables transformation from an algebraic style of programming resembling mathematical specification of numeric programs to an updating style in which function arguments are overwritten in order to reuse memory allocated to large matrices \[2,13\].

Stratego \[13,15\] is a domain-specific language for the specification of program transformation systems based on the paradigm of rewriting strategies. Stratego separates the specification of basic transformation rules from that of the strategies by means of which they are applied. Strategies that control the application of transformation rules can be programmed using a small set of primitive strategy combinators. These combinators support the definition of very generic patterns of control, allowing strategies and rules to be composed as necessary to achieve various program transformations. This abstract programming style leads to concise and reusable specifications of program transformation systems. However, due to their genericity, some strategies do not have enough information to perform their tasks efficiently, even though specializations of those strategies could be implemented efficiently.

In this paper we develop a fusion technique for Stratego programs that specializes the generic innermost reduction strategy to specific sets of rules. This optimization supports abstract programming while obtaining the efficiency of hand-written specializations. The optimization is implemented in Stratego itself by means of local transformations. A local transformation is one that is applied to a selected part of a program under the control of a strategy.

In conventional program optimization, transformations are applied throughout a program. In optimizing imperative programs, complex transformations are applied to entire programs \[12\]. In the style of compilation by transformation \[3\] — as applied, for example, in the Glasgow Haskell Compiler \[14\] — a large number of small, almost trivial program transformations are applied throughout a program to achieve large-scale optimization by accumulating small program changes. The style of optimization that we develop in this paper is a combination of these ideas: Combine a number of small transformation steps using strategies that will apply them to specific parts of a program to achieve the effects of complex transformations. Because the transformations are local, special knowledge about the subject program at the point of appli-
cation can be used. This allows the application of rules that would not be otherwise applicable.

The remainder of this paper is organized as follows. In Section 2 we explain the basics of Stratego and introduce the generic Stratego specification of innermost reduction. We present an optimized version of this strategy in Section 3. In Section 4 we show how the optimized specification of innermost can be derived from the original specification. Section 5 presents the Stratego implementation of the optimization rules from Section 4. Section 6 concludes.

2 A Generic Specification of Innermost Reduction

Stratego is a language for specifying program transformations. A key design choice of the language is the separation of logic and control. The logic of program transformations is captured by rewrite rules, while rewriting strategies control the application of those rules.

In this section we describe the elements of Stratego that are relevant for this paper. We illustrate them with a small application which simplifies expressions over natural numbers with addition using a generic specification of innermost reduction. A complete description of Stratego, including a formal semantics, is given in [16].

In Stratego, programs to be transformed are expressed as first-order terms. Signatures describe the structure of terms. A term over a signature $S$ is either a nullary constructor $C$ from $S$ or the application $C(t_1, \ldots, t_n)$ of an $n$-ary constructor $C$ from $S$ to terms $t_i$ over $S$. For example, Zero, Succ(Zero), and Plus(Succ(Zero),Zero) are terms over the signature in Figure 1.

2.1 Rewrite Rules

Rewrite rules express basic transformations on terms. A rewrite rule has the form $L : l \rightarrow r$, where $L$ is the label of the rule, and the term patterns $l$ and $r$ are its left-hand side and right-hand side, respectively. A term pattern is either a variable, a nullary constructor $C$, or the application $C(p_1, \ldots, p_n)$.

```
module peano
signature
  sorts Nat
  constructors
    Zero : Nat
    Succ : Nat -> Nat
    Plus : Nat * Nat -> Nat
rules
  A : Plus(Zero, x) -> x
  B : Plus(Succ(x), y) -> Succ(Plus(x, y))
```

Fig. 1. An example Stratego module with signature and rewrite rules.
of an n-ary constructor \( C \) to term patterns \( \pi \). For example, Figure 1 shows rewrite rules A and B that simplify sums of natural numbers. As suggested there, Stratego provides a simple module structure that allows modules to import other modules.

A rule \( L: l \rightarrow r \) applies to a (ground) term \( t \) when the pattern \( l \) matches \( t \), i.e., when \( l \) has the same top-level structure as \( t \). Applying \( L \) to \( t \) has the effect of transforming \( t \) to the term obtained by replacing the variables in \( r \) with the subterms of \( t \) to which they correspond. For example, rule B transforms the term \( \text{Plus}(\text{Succ}(\text{Zero}),\text{Succ}(\text{Zero})) \) to the term \( \text{Succ}(\text{Plus}(\text{Zero},\text{Succ}(\text{Zero}))) \), where \( x \) corresponds to \( \text{Zero} \) and \( y \) corresponds to \( \text{Succ}(\text{Zero}) \).

In the normal interpretation of term rewriting, terms are normalized by exhaustively applying rewrite rules to a term and its subterms until no further applications are possible. The term \( \text{Plus}(\text{Succ}(\text{Zero}),\text{Zero}) \), for instance, normalizes to the term \( \text{Succ}(\text{Zero}) \) under rules A and B. But because normalizing a term with respect to all rules in a specification is not always desirable, and because rewrite systems need not be confluent or terminating, more careful control is often necessary. A common solution is to introduce additional constructors into signatures and then use them to encode control by means of additional rules which specify where and in what order the original rules are to be applied. Programmable rewriting strategies provide an alternative mechanism for achieving such control while avoiding the introduction of new constructors or rules.

2.2 Combining Rules with Strategies

Figures 2 and 3 illustrate how strategies can be used to control rewriting. Figure 2 gives a generic definition of the notion of innermost normalization under some transformation \( s \). The innermost strategy can be instantiated with any selection of rules to achieve normalization of terms under those rules. For instance, in Figure 3 the strategy main is defined to normalize \( \text{Nat} \) terms using the innermost strategy instantiated with rules A and B. In general, transformation rules and reduction strategies can be defined independently and can be combined in various ways. A different selection of rules can be made, or the rules can be applied using a different strategy. Not all rules in a specification are required to participate in a specific normalization. In this way, it is possible in Stratego to develop a library of valid transformation rules and apply them in various transformations as needed.

2.3 Rewriting Strategies

A rewriting strategy is a program that transforms terms or fails at doing so. In the case of success, the result is a transformed term or the original term. In the case of failure, there is no result.

Rewrite rules are just strategies which apply transformations to the roots
of terms. Strategies can be combined into more complex strategies by means of Stratego’s strategy operators. The identity strategy \texttt{id} always succeeds and leaves its subject term unchanged. The failure strategy \texttt{fail} always fails. The sequential composition \texttt{s1 ; s2} of strategies \texttt{s1} and \texttt{s2} first attempts to apply \texttt{s1} to the subject term. If that succeeds, it applies \texttt{s2} to the result; otherwise it fails. The non-deterministic choice \texttt{s1 + s2} of strategies \texttt{s1} and \texttt{s2} attempts to apply either \texttt{s1} or \texttt{s2} to the subject term, but in an unspecified order. It succeeds if either \texttt{s1} or \texttt{s2} succeeds, and fails otherwise. The deterministic choice \texttt{s1 <+ s2} of strategies \texttt{s1} and \texttt{s2} first attempts to apply \texttt{s1} to the subject term. If \texttt{s1} fails, then it attempts to apply \texttt{s2} to the subject term. If \texttt{s1} and \texttt{s2} both fail, then it fails as well. The recursive closure \texttt{rec x(s)} of a strategy \texttt{s} attempts to apply to the subject term the strategy obtained by replacing each occurrence of the variable \texttt{x} in \texttt{s} by the strategy \texttt{rec x(s)}.

A strategy definition \texttt{f(x1,...,xn) = s} introduces a new strategy operator \texttt{f} parameterized with strategies \texttt{x1,...,xn} and having body \texttt{s}. Such definitions cannot refer (directly or indirectly) to the operator being defined. Instead, all recursion must be expressed explicitly by means of the recursion operator \texttt{rec}.

### 2.4 Term Traversal

The strategy combinators just described combine strategies which apply transformation rules to the roots of their subject terms. In order to apply a rule at an internal site of a term (i.e., to a subterm), it is necessary to traverse the term. Stratego defines several primitive operators which expose the direct subterms of a constructor application. These can be combined with the operators described above to define a wide variety of complete term traversals. For the purposes of this paper we restrict the discussion of traversal operators to congruence operators and the \texttt{all} operator.

Congruence operators provide one mechanism for term traversal in Stratego. For each constructor \texttt{C} there is a corresponding congruence operator.
If $C$ is an $n$-ary constructor, then the corresponding congruence operator defines the strategy $C(s_1,\ldots,s_n)$, which applies only to terms of the form $C(t_1,\ldots,t_n)$ resulting in $C(t_1',\ldots,t_n')$, if each $s_i$ successfully applies to $t_i$ resulting in $t_i'$. For example, the congruence $\text{Plus}(s_1,s_2)$ applies only to $\text{Plus}$ terms, and it works by applying $s_1$ to the first summand and $s_2$ to the second, producing $\text{Plus}(t_1',t_2')$. If the application of $s_i$ to $t_i$ fails for any $i$, then the application of $C(s_1,\ldots,s_n)$ to $C(t_1,\ldots,t_n)$ also fails.

The operator $\text{all}(s)$ applies $s$ to each of the direct subterms $t_i$ of a constructor application $C(t_1,\ldots,t_n)$. It succeeds if and only if all applications to the direct subterms succeed. The resulting term is the constructor application $C(t_1',\ldots,t_n')$ where the $t_i'$ are the results obtained by applying $s$ to the terms $t_i$. Note that $\text{all}(s)$ is the identity on constants, i.e., on constructor applications without children. An example of the use of $\text{all}$ appears in the strategy $\text{bottomup}$ in Figure 2. The strategy expression $\text{rec} \ x(\text{all}(x); \ s)$ specifies that the strategy is first applied recursively to all direct subterms of a term, and, thereby, to all of its subterms. If that succeeds, then the argument strategy $s$ is applied to the resulting term. This definition of $\text{bottomup}$ captures the generic notion of a post-order traversal over a term.

The $\text{innermost}$ strategy in Figure 2 is defined using $\text{bottomup}$. The strategy $\text{innermost}(s)$ performs a bottomup traversal over a term. At each subterm it calls the strategy $\text{red}(s)$ to reduce that subterm. This means that before $\text{red}(s)$ is applied to a term, all its subterms are normalized with respect to $s$. The strategy $\text{red}(s)$ then applies the transformation $s$ to the subject term. If that succeeds, then the result of the transformation is further reduced by invoking a $\text{bottomup}$ traversal, which recursively calls the $\text{red}(s)$ transformation at each subterm. If not, then this entails that the subject term is in normal form, and so $\text{red}(s)$ succeeds with $\text{id}$. The strategy $\text{innermost}$ thus captures the notion of parallel innermost reduction. Note, however, that other specifications of $\text{innermost}$ are possible since Stratego representations of strategies are not, in general, unique.

### 3 An Optimized Specification of Innermost Reduction

Inspection of the specification for the $\text{innermost}$ strategy in Figure 2 reveals an inefficiency resulting from the up-and-down way in which it traverses terms. The difficulty is that subterms which have already been normalized may be reconsidered for normalization a number of times. Consider, for example, rule $B$ from Figure 1. Because of the way $\text{innermost}$ is defined, the subterms of the term matched by the left-hand side of $B$ in an application of $\text{main}$ are already in normal form before application of the rule. In particular, the terms matching the variables $x$ and $y$ are in normal form. However, after constructing the right-hand side $\text{Succ}(\text{Plus}(x,y))$, the terms $x$ and $y$ are completely renormalized by the occurrence of $\text{bottomup}$ in the called strategy $\text{red}$. Renormalization entails that these terms are completely traversed and

---

[1] Figure 1 and Figure 2 are not provided in the text, but they are assumed to be relevant to the discussion.
that the rules are tried at each subterm. Since the terms are in normal form, no actual transformation is done, of course.

In the specific case of innermost normalization with the rules $A$ and $B$, a more efficient definition is the one in Figure 4. This definition completely avoids renormalization — as is easily seen once we introduce the Stratego constructs it uses in Section 4 — but there are (at least) two problems with optimizations such as this one. First, it can be quite difficult to optimize strategies by hand. Hand optimization is error-prone, especially when performed on specifications of any reasonable size. Second, rules and strategies tend to be intermingled in optimized programs. This inhibits both reuse of the rules with other strategies and their reuse in combinations other than those which have been “hard wired” into the optimized strategy. For these reasons, automatic transformation of modular specifications into optimized versions is desirable.

In the next section we justify our optimization of \texttt{innermost} by showing how the optimized specification in Figure 4 can be derived automatically from that in Figure 2. We demonstrate the technique by applying it to the specific program \texttt{innermost(A+B)}, but it optimizes all uses of \texttt{innermost} applied to any selection of rules equally well.

4 Derivation

In this section we show how the optimized implementation of Figure 4 can be derived from the strategy \texttt{innermost(A + B)} by systematic transformation. In the next section we will formalize in Stratego the transformation rules we use and will develop a strategy for automatically applying them in the correct order.

The goal of the derivation is to fuse the occurrence of \texttt{bottomup} appearing in the definition of the strategy \texttt{red} called by \texttt{innermost} with the right-hand sides of the rules $A$ and $B$. This avoids renormalizing the terms to which variables from the left-hand sides of these rules are bound. To achieve this, we first desugar the rules and then inline (unfold) definitions in order to arrive at a single expression containing the complete specification of the \texttt{innermost} strategy. The \texttt{bottomup} strategy can then be distributed over the right-hand sides of the rules to which \texttt{innermost} applies; $A$ and $B$ in our running example.
4.1 Desugaring Rules

In Stratego, rules are not primitives. Instead they are expressed in terms of primitives for matching and building terms. The strategy \( ?t \) matches the subject term against the term pattern \( t \). The strategy \( !t \) replaces the subject term with the term constructed by instantiating the variables in the term pattern \( t \) with their current bindings. The construct \( \{xs:s\} \) delimits the scope of the variables in the strategy \( s \). Thus, a rule \( L: l \rightarrow r \) is just syntactic sugar for \( L = \{x_1, \ldots, x_n: ?l; !r\} \), where the \( x_i \) are the variables occurring in the rule. The example rules from module \texttt{peano} in Figure 1 thus reduce to

\[
A = \{x: ?\text{Plus}(\text{Zero}, x); !x\} \\
B = \{x,y: ?\text{Plus}(\text{Succ}(x), y); !\text{Succ}(\text{Plus}(x, y))\}
\]

4.2 Inlining Definitions

The first step of the derivation consists in \textit{inlining} definitions, i.e., in replacing each call to a strategy by the body of its definition. If \( f(s_1, \ldots, s_n) = s \) is the definition of strategy operator \( f \), then a call \( f(s_1, \ldots, s_n) \) to that operator can be replaced by \( s[s_1/x_1, \ldots, s_n/x_n] \), i.e., by the strategy obtained by replacing the formal parameters of the body of \( f \) by its actual arguments. In the case of the \texttt{main} strategy in module \texttt{apply-peano} in Figure 3, inlining gives

(1) \texttt{innermost(A + B)}

By the definition of \texttt{innermost} this expands to

(2) \texttt{bottomup(red(A + B))}

By the definition of \texttt{red}, this in turn gives

(3) \texttt{bottomup(rec r((A + B); bottomup(r) <+ id))}

Finally, inlining the definitions of rules \( A \) and \( B \) gives

(4) \texttt{bottomup(rec r(}

\[
\begin{aligned}
&\{x: ?\text{Plus}(\text{Zero}, x); !x\} \\
+ \{x,y: ?\text{Plus}(\text{Succ}(x), y); !\text{Succ}(\text{Plus}(x, y))\} \\
&); \text{bottomup}(r) \\
& <+ \text{id})
\end{aligned}
\]

4.3 Sequential Composition over Choice

In the next step of the derivation we right distribute the \texttt{bottomup} strategy over the nondeterministic choice strategy using the rule

\[
(x + y); z \rightarrow (x; z) + (y; z)
\]

This rule is not valid for all strategy expressions. Consider a term \( t \) for which \( x \) and \( y \) both succeed, \( (x;z) \) fails, and \( (y;z) \) succeeds. Then \( (x + y); z \) will fail if application of \( x \) is attempted. By contrast, \( (x;z) + (y;z) \) will always
succeed since \((y;z)\) does. It is, however, the case that the rule does hold whenever \(z\) is guaranteed to succeed; in this situation, the success or failure of both sides of the rule is determined wholly by the success or failure of \(x\) and \(y\).

Since \(\text{id}\) always succeeds, \(\mathbf{r}\) in the recursive strategy (4) is guaranteed to succeed as well. Thus, \(\text{bottomup}(\mathbf{r})\) is guaranteed to succeed, and so right distribution of \(\text{bottomup}(\mathbf{r})\) according to the rule is valid. This gives

\[
\text{(5) bottomup}(\text{rec } \mathbf{r}(\{x: \text{?Plus}(\text{Zero}, x); \text{!x}; \text{bottomup}(\mathbf{r})
+ \{x, y: \text{?Plus}(\text{Succ}(x), y); \text{!Succ}(\text{Plus}(x, y)); \text{bottomup}(\mathbf{r})
\}) \leftarrow \text{id}))
\]

4.4 Sequential Composition over Scope

Next, in order to apply \(\text{bottomup}(\mathbf{r})\) to the right-hand sides of the rules we need to bring it under the scope of the rules by applying the transformation

\[
\{\text{xs: s1}; \text{s2} \rightarrow \{\text{xs: s1}; \text{s2}\}
\]

This rule is valid if the variables in \(\text{xs}\) are not free in \(\text{s2}\). Its application transforms (5) into

\[
\text{(6) bottomup}(\text{rec } \mathbf{r}(\{x: \text{?Plus}(\text{Zero}, x); \text{!x}; \text{bottomup}(\mathbf{r})
+ \{x, y: \text{?Plus}(\text{Succ}(x), y); \text{!Succ}(\text{Plus}(x, y)); \text{bottomup}(\mathbf{r})
\}) \leftarrow \text{id}))
\]

4.5 Strategy Application

We can now apply \(\text{bottomup}(\mathbf{r})\) to the term built in the right-hand side of each rule. Using the notation \(<\mathbf{s}> t\) to denote \(!t; \mathbf{s}\), i.e., to denote application of the strategy \(\mathbf{s}\) to the instance of \(t\) determined by the current bindings, we get

\[
\text{(7) bottomup}(\text{rec } \mathbf{r}(\{x: \text{?Plus}(\text{Zero}, x); <\text{bottomup}(\mathbf{r})> x
+ \{x, y: \text{?Plus}(\text{Succ}(x), y); <\text{bottomup}(\mathbf{r})> \text{Succ}(\text{Plus}(x, y))
\}) \leftarrow \text{id}))
\]

4.6 Distribution of bottomup

The application of \(\text{bottomup}(\mathbf{r})\) to a constructor application leads to the following derivation:

\[
<\text{bottomup}(\mathbf{r})> \text{C}(t_1, \ldots, t_n)
= \{\text{definition of bottomup}\}
<\text{rec } x(\text{all}(x); \mathbf{r}) > \text{C}(t_1, \ldots, t_n)
= \{\text{recursion}\}
<\text{all}(\text{rec } x(\text{all}(x); \mathbf{r}); \mathbf{r}) > \text{C}(t_1, \ldots, t_n)
\]
By repeatedly applying this rule, bottomup\(\text{r}\) is distributed over the term constructions in the right-hand sides until variables are encountered. This gives

\(\text{(8) bottomup(r (}
\begin{array}{l}
\{x : ?\text{Plus}(\text{Zero, } x); <\text{bottomup(r)}> x\}
+ \{x, y: ?\text{Plus}(\text{Succ}(x), y);
\quad <r> \text{Succ}(<r> \text{Plus(<bottomup(r)> x,<bottomup(r)> y)))}
\end{array})\ <+ \text{id})\)

### 4.7 Avoiding Renormalization

Finally, we use the observation that

\(<\text{bottomup(r)> v -> v}\)

if \(v\) is a variable originating in the left-hand side of a rule. That is, if \(v_s\) contains all variables occurring in \(l\), \(v\) is in \(v_s\), and \(\{v_s: ?l; !r\}\) is a strategy, then occurrences of \(<\text{bottomup(r)>v}\) in \(r\) can be replaced by \(v\) itself. This observation is valid because terms matching variables from the left-hand side of a rule are already in normal form.

Although this observation relies on non-local information, it does give rise to a transformation which is local in the sense that it is applied only within a single strategy, i.e., that is applied (locally) to a selected part of a program under the control of a strategy. Using it, we arrive at the desired optimized version of \(\text{innermost}(\text{A+B})\):

\(\text{(9) bottomup(r (}
\begin{array}{l}
\{x: ?\text{Plus}(\text{Zero, } x); !x\}
+ \{x, y: ?\text{Plus}(\text{Succ}(x), y); <r> \text{Succ(<r> \text{Plus(x, y))})}
\end{array})\ <+ \text{id})\)

### 5 Implementation

In this section we show how the rules used in the derivation in Section 4 can be implemented in Stratego. We start by defining the abstract syntax of Stratego programs. We then add overlays to abstract over specific patterns in the abstract syntax that occur often in the rules. Next, we formalize the rules used in the derivation as Stratego rules. Finally, we combine these rules into a strategy that optimizes occurrences of the \(\text{innermost}\) strategy in Stratego specifications. The optimization works for all strategies of the form \(\text{innermost} \text{(R1 + ... + Rn)}\) with arbitrary rules \(Ri\).
5.1 Abstract Syntax

Figure 5 defines the signature of the abstract syntax of terms and strategy expressions in Stratego. The signature has been reduced to those constructs that are relevant to optimizing innermost. The term

\[(10) \text{Scope}(["x"], \text{Seq}(\text{Match}(\text{Op}(\text{Plus},[\text{Op}(\text{Zero},[]),\text{Var}("x")]), \text{Build}(\text{Var}("x"))))}\]

over this signature is the abstract syntax representation for the body of rule A from Figure 1 after desugaring, and

\[(11) \text{Scope}(["x","y"], \text{Seq}(\text{Match}(\text{Op}(\text{Plus},[\text{Op}(\text{Succ},[\text{Var}("x")]),\text{Var}("y"))]), \text{Build}(\text{Op}(\text{Succ},[\text{Op}(\text{Plus},[\text{Var}("x"),\text{Var}("y")])]))))}\]

is the representation for rule B.

As suggested by this signature, Stratego supports the built-in data type String. Syntactic sugar for lists in the form \([t_1,\ldots,t_n]\) is also provided.

---

module stratego
signature
sorts Term
constructors
   Var : String -> Term
   Op : String * List(Term) -> Term
sorts SVar Strat SDef
constructors
   Id : Strat
   Fail : Strat
   Seq : Strat * Strat -> Strat
   Choice : Strat * Strat -> Strat
   LChoice : Strat * Strat -> Strat
   SVar : String -> SVar
   Rec : String * Strat -> Strat
   SDef : String * List(String) * Strat -> SDef
   Call : SVar * List(Strat) -> Strat
   All : Strat -> Strat
   Match : Term -> Strat
   Build : Term -> Strat
   Scope : List(String) * Strat -> Strat
   Where : Strat -> Strat

Fig. 5. Simplified abstract syntax of Stratego programs.
5.2 Patterns in Abstract Syntax

We want to optimize certain specific patterns of strategy expressions. Since we do not want to rely on the names chosen for those patterns by the specification writer, we need to be able to recognize the structure of patterns. Because encoding patterns using abstract syntax expressions can lead to large unmanageable terms, we use the Stratego overlay mechanism to abstract over them.

An overlay gives a name (pseudo-constructor) to a complex term pattern. The pseudo-constructor can then be used as an ordinary constructor in matching and building terms. Overlays can use other overlays in their definitions, but cannot be recursive. Overlays can be thought of as term macros. Using overlays, we can write concise transformation rules involving complex term patterns. Module strategy-patterns in Figure 6 defines overlays for the abstract syntax patterns corresponding to the strategies innermost, bottomup, and red from the example in Figure 2. Thus, the overlay Do("f") is an abbreviation of the term Call(SVar("f"),[]). The “extra” parameters in Figure 6 correspond to bound variables from Figure 2.

5.3 Transformation Rules

Figure 7 defines the rules that were used in the derivation in Section 4. The first six rules are distribution rules for sequential composition over other operators. The right distribution rules for sequential composition over deterministic and non-deterministic choice are parameterized with strategies that decide whether or not the strategy expression to be distributed is guaranteed to succeed. The AssociateR rule associates composition to the right. The IntroduceApp rule defines the transformation !t; s -> <s> t. Finally, rule BottomupOverConstructor distributes Bottomup over constructor application. The rule uses the map strategy operator to distribute the application of Bottomup over the list of arguments of the constructor.

As Figure 7 suggests, Stratego rules can have conditions which are introduced using the keyword where. Conditional rules apply only if the conditions in their where clauses succeed. In addition, the notation \r\ converts a rule r into a strategy. The argument to map in BottomupOverConstructor is thus the strategy corresponding to the local rule that transforms a term t into an application of the same instance of Bottomup to t.
module fusion-rules
imports stratego
rules
  SeqOverChoiceL : Seq(x, Choice(y, z)) -> Choice(Seq(x, y), Seq(x, z))

  SeqOverLChoiceL : Seq(x, LChoice(y, z)) -> LChoice(Seq(x, y), Seq(x, z))

  SeqOverChoiceR(succ) : Seq(Choice(x, y), z) -> Choice(Seq(x, z), Seq(y, z))
  where <succ> z

  SeqOverLChoiceR(succ) : Seq(LChoice(x, y), z) -> LChoice(Seq(x, z), Seq(y, z))
  where <succ> z

  SeqOverScopeR : Seq(Scope(xs, s1), s2) -> Scope(xs, Seq(s1, s2))

  SeqOverScopeL : Seq(s1, Scope(xs, s2)) -> Scope(xs, Seq(s1, s2))

  AssociateR : Seq(Seq(x, y), z) -> Seq(x, Seq(y, z))

  IntroduceApp : Seq(Build(t), s) -> Build(App(s, t))

  BottomupOverConstructor :
    App(Bottomup(x, s), Op(c, ts)) ->
    App(s, Op(c, <map(\ t -> App(Bottomup(x, s), t)\ )> ts))

Fig. 7. Distribution and association rules.

5.4 Strategy

Figure 8 defines the strategy fusion that combines the rules in Figure 7 into a strategy for optimizing occurrences of the innermost strategy. It is assumed that desugaring and inlining have already been performed prior to application of fusion. These transformations are handled automatically by the Stratego compiler.

The fusion strategy sequences four constituent strategies. First, an occurrence of the innermost strategy is recognized using the congruence operator corresponding to the Innermost overlay. The IntroduceMark rule applies a mark to the choice of rules to be used in the innermost normalization of terms, and the strategy propagate-mark then propagates the mark to the argument
module fusion-strategy
imports strategy-patterns fusion-rules
signature
  constructors
    Mark : Strat

strategies

fusion =
    Innermost(IntroduceMark,id,?r,id);
    propagate-mark;
    fuse-with-bottomup(?Bottomup(_, Do(r)));
    alltd(BottomupToVarIsId(?Do(r)));

propagate-mark =
    innermost(SeqOverChoiceL + SeqOverLChoiceL + SeqOverScopeL)

fuse-with-bottomup(succ) =
    innermost(SeqOverChoiceR(succ) + SeqOverLChoiceR(succ)
               + SeqOverScopeR + AssociateR + IntroduceApp
               + BottomupOverConstructor)

rules

IntroduceMark : s -> Seq(Mark, s)

BottomupToVarIsId(isr) :
    Seq(Mark, Seq(Match(lhs), Build(rhs))) ->
    Seq(Match(lhs), Build(rhs'))
where <tvars> lhs => vs;
    <alltd(App(Bottomup(_,r), Var(v)) -> Var(v)
           where <fetch(?v)> vs; <isr> r \} rhs => rhs'

Fig. 8. Fusion strategy.

rules. The propagated marks make it possible to distinguish normalizing rules from local rules in the normalizing strategy. Note that it is an additional constructor that is used to convey information from one transformation to the next. Although strategies often make it possible to avoid additional constructors, they are sometimes still needed.

Using the pseudoconstructors Innermost, Bottomup, and Red to enhance readability, we can express the result of applying the Innermost congruence of fusion to the abstract syntax representation for innermost(A+B). In this notation, innermost(A+B) is abbreviated

(12) Innermost(Choice(alpha,beta),p,w,z)

where alpha and beta are the abstract syntax representations for rules A and
B, respectively, given in (10) and (11), and p, w, and z are new auxiliary variables corresponding to the bound variables in Figure 4. Applying the specified Innermost overlay to (12) yields

(13) Innermost(Seq(Mark,Choice(alpha,beta)),p,w,z)

This corresponds to the strategy in (4). Applying propagate-mark to (13) then gives

(14) Innermost(Choice(Seq(Mark,alpha),Seq(Mark,beta)),p,w,z)

which corresponds to the strategy in (5).

Next, the strategy fuse-with-bottomup distributes trailing occurrences of Bottomup over choice, scope, build, and constructors. At this point the right-hand sides of rules have the form of expression (8) in the previous section. In particular, applying fuse-with-bottomup to (14) gives

(15) Bottomup(p,Rec(w,LChoice(Choice(alpha1,beta1),Id)))

where alpha1 is

(16) Scope(["x"],Seq(Mark,
      Seq(Match(Op("Plus",[Op("Zero",[]),Var("x")])),
      Build(App(Bottomup(z,Do(w)),Var("x")))))

and beta1 is

(17) Scope(["x","y"],Seq(Mark,
      Seq(Match(Op("Plus",[Op("Succ",[Var("x")],Var("y")])),
      Build(App(Do(w),
      Op("Succ",
      [App(Do(w),
      Op("Plus",
      [App(Bottomup(z,Do(w)),Var("x")],
      App(Bottomup(z,Do(w)),Var("y"))]
      ))))))))

Finally, BottomupToVarIsId removes the applications of Bottomup(_,r) to variables in the right-hand sides of marked rules provided these also occur in their left-hand sides. The first local rule of BottomupToVarIsId uses tvars to record those variables occurring in the left-hand side of a normalizing rule. The second then removes applications of Bottomup(_,r) to occurrences of these variables in the right-hand side of the normalizing rule by means of a local traversal of that right-hand side (rhs).

The notation <s> t => t' in the conditions of the rule BottomupToVarIsId abbreviates !t; s; ?t'. The traversal strategy alltd used there is defined as

alltd(s) = rec x(s <+ all(x))

It performs all outermost applications of s in the subject term by first attempting to apply s to the root of the subject term and, if this fails, recursively
attempting to apply \textit{s} to each child. The argument to \texttt{alltd} in the rule \texttt{BottomupToVarIsId} is the strategy derived from the local rule

\begin{verbatim}
App(Bottomup(_,r), Var(v)) -> Var(v)
where <fetch(?v)> vs; <isr> r
\end{verbatim}

which removes the application of \texttt{Bottomup(_,r)} to \texttt{Var(v)}. It uses the condition \texttt{<fetch(?v)> vs} to determine whether or not the variable \texttt{v} to which \texttt{Bottomup(_,r)} is applied appears in the left-hand side of a marked rule, i.e., is in the list of variables \texttt{vs}. The strategy \texttt{isr} is passed to the rule by the fusion strategy, and indicates whether or not \texttt{r} is indeed the recursion variable. This ensures that only the desired applications of \texttt{Bottomup(_,r)} are removed.

Applying the \texttt{alltd} traversal specified in fusion to (15), for example, gives

\begin{verbatim}
(18) Bottomup(p,Rec(w,LChoice(Choice(alpha2,beta2),Id)))
\end{verbatim}

where \texttt{alpha2} is

\begin{verbatim}
(19) Scope(["x"],
    Seq(Match(Op("Plus",[Op("Zero",[]),Var("x")]),
              Build(Var("x")))))
\end{verbatim}

and \texttt{beta2} is

\begin{verbatim}
(20) Scope(["x","y"],
    Seq(Match(Op("Plus",[Op("Succ",[Var("x")]),Var("y")]),
              Build(App(Do(w),
                        Op("Succ",
                           [App(Do(w),
                             Op("Plus",[Var("x"),Var("y")]))])))))))
\end{verbatim}

This is the final result of applying \textit{fusion} to the abstract syntax representation of \texttt{innermost(A+B)}. It corresponds to the strategy in (9).

6 Concluding Remarks

In this paper we have shown how local transformations can be used to fuse logic and control in optimizing abstract programs. Strategies play two important roles in our approach. First, they appear as abstract programming devices that are subject to optimization. Second, together with local transformation rules, they provide a language in which automatic optimizations can be specified in an elegant manner. Strategy-based optimization can thus be used to reduce inefficiencies associated with the genericity of strategies as programming tools.

The optimization strategy presented in this paper is included as an experimental optimization phase in the Stratego compiler (version 0.5.4). The optimization still has some limitations. Only strategies of the form \texttt{innermost(R1 + \ldots + Rn)}, where the \texttt{Ri} are rules, are optimized. Strategies such as
innermost规则1 + 规则2），其中策略规则1和规则2定义为规则1 = R11 + ... + R1k 和规则2 = R21 + ... + R2l，如果没有处理好因为编译器目前不内嵌这样的定义。这需要对编译器进行一般化。此外，规则与条件引入新变量是处理的，但绑定到新引入变量的项是重新归一化的。基于对条件的分析，这在某些情况下可以避免。


小型局部变换在[14]中被称作“卑微变换”。这些变换在编译器中被广泛用于优化基于编译-转换的编译器[8,9,1,13]。它们也在某种程度上被大多数编译器使用，尽管在实现中可能不被认为是可识别的重写规则。


最后，需要衡量innermost融合和相关融合技术的实现。
Acknowledgments

Patricia Johann is supported in part by the National Science Foundation under grant CCR-9900510. We thank Simon Peyton Jones, Andrew Tolmach, Roy Dyckhoff and the WRS referees for their remarks on a previous version of this paper.

References


