Fusing Logic and Control with Local Transformations: An Example Optimization

Patricia Johann(1) and Eelco Visser(2)

(1) Mathematics Department, Bates College,
Lewiston, Maine, USA. pjohann@bates.edu
(2) Institute of Information and Computing Sciences,
Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht,
The Netherlands. visser@acm.org

Abstract. Abstract programming supports separation of concerns, and leads to reduced code size and increased reusability of code. Its main disadvantage is computational overhead. Fusion techniques can be used to combine the reusability of abstract programs with the efficiency of specialized programs. In this paper we illustrate how rewriting strategies can be used for the generic definition of program transformation components. In particular, 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 functions [7, 10, 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, 4].

Stratego [15, 16] 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 only within a single strategy, i.e., that is applied (locally) 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 [11]. In the style of compilation by transformation [3] — as applied, for example, in the Glasgow Haskell Compiler [13] — 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 application 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 contains the Stratego implementation of the optimization rules from Section 4. Section 6 concludes.

2 A Generic Specification of Innermost

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

```
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.

transformations is captured by rewrite rules, while rewriting strategies control the application of rewrite rules.

In this section we explain 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(t1,...,tn)` of an `n`-ary constructor `c` from `S` to terms `ti` 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 -> 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(p1,...,pn)` 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 -> r` applies to a 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 `Plus(Succ(Zero),Succ(Zero))` to `Succ(Plus(Zero,Succ(Zero)))`. Here `x` corresponds to `Zero` and `y` corresponds to `Succ(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 `Plus(Succ(Zero),Zero)`, for instance, normalizes to the term `Succ(Zero)` under rules `A` and `B`. But because normalizing

```
module apply-peano
imports innermost peano
strategies
  main = innermost(A + B)
```

Fig. 2. Using Peano rules.

```
module innermost
strategies
  innermost(s) = bottomup(red(s))
  bottomup(s)  = rec r(all(r); s)
  red(s)       = rec x(s; bottomup(x) <+ id)
```

Fig. 3. Generic traversal strategies.

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 3 gives a generic definition of the notion of innermost normalization under some transformation `s`. The innermost strategy can be instantiated with any choice of rules to achieve normalization of terms under those rules. For instance, in Figure 2 the strategy `main` is defined to normalize `Nat` terms using the 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. 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 `id` always succeeds and leaves its subject term unchanged. The *failure* strategy `fail` always fails. The *sequential composition* `s1 ; s2` of strategies `s1` and `s2` first attempts to apply `s1` to the subject term. If that succeeds, it applies `s2` to the result; otherwise it fails. The *non-deterministic choice* `s1 + s2` of strategies `s1` and `s2` attempts to apply either `s1` or `s2` to the subject term, but in an unspecified order. It succeeds if either `s1` or `s2` succeeds, and fails otherwise. The *deterministic choice* `s1 <+ s2` of strategies `s1` and `s2` first attempts to apply `s1` to the subject term. If `s1` fails, then it attempts to apply `s2` to the subject term. If `s1` and `s2` both fail, then it fails as well. The *recursive closure* `rec x(s)` of a strategy `s` attempts to apply to the subject term the strategy obtained by replacing each occurrence of the variable `x` in `s` by the strategy `rec x(s)`.

A strategy definition `f(x1,...,xn) = s` introduces a new strategy operator `f` parameterized with strategies `x1,...,xn` and having body `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 `rec`.

2.4 Generic 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 `all` operator.

Congruence operators provide one mechanism for term traversal in Stratego. If `c` is an `n`-ary constructor, then the congruence `c(s1,...,sn)` is the strategy that applies only to terms of the form `c(t1,...,tn)`, and works by applying the strategies `si` to the terms `ti`. For example, the congruence `Plus(s1,s2)` applies only to `Plus` terms, and it works by applying `s1` to the first summand and `s2` to the second. If the application of `si` to `ti` fails for any `i`, then the application of `c(s1,...,sn)` to `c(t1,...,tn)` also fails.

The operator `all(s)` applies `s` to all direct subterms `ti` of a constructor application `C(t1,...,tn)`. It succeeds if and only if all applications to the direct subterms succeed. The resulting term is the constructor application `C(t1',...,tn')` where the `ti'` are the results obtained by applying `s` to the terms `ti`. Note that `all(s)` is the identity on constants, i.e., on constructor applications without children. An example of the use of `all` appears in the strategy `bottomup` in Figure 3. The strategy expression `rec x(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 `bottomup` captures the generic notion of a post-order traversal over a term.

The innermost strategy in Figure 3 is defined using `bottomup`. The strategy `innermost(s)` does a bottomup traversal over a term. At each subterm it calls the strategy `red(s)` to reduce that subterm. This means that before `red(s)` is applied to a term, all its subterms are normalized with respect to `s`. The strategy `red(s)` then applies the transformation `s` to the subject term. If that succeeds, then the result of the transformation is further normalized. If not, then this entails that the subject term is in normal form, and so `red(s)` succeeds with `id`. The strategy `innermost` thus captures the notion of parallel innermost reduction. Note, however, that other specifications of `innermost` are possible since Stratego representations of strategies are not, in general, unique.

3 An Optimized Specification of Innermost

Inspection of the specification for the innermost strategy in Figure 3 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 `innermost` is defined, the subterms of the term matched by the left-hand side of `B` 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 `Succ(Plus(x,y))`, the terms `x` and `y` are completely renormalized by the occurrence of `bottomup` in the called strategy `red`. Renormalization entails that these terms are completely traversed and 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 `innermost` by showing how the optimized specification in Figure 4 can be derived automatically from that in

```
module apply-peano
strategies
  main =
    bottomup(rec r(
      ( {x: ?Plus(Zero, x); !x}
      + {x, y: ?Plus(Succ(x), y); <r> Succ(<r> Plus(x, y))}
      ) <+ id))
```

Fig. 4. Optimized strategy.

Figure 3. We demonstrate the technique by applying it to the specific program `innermost(A+B)`, but it optimizes all uses of `innermost` applied to a choice of rules equally well.

4 Derivation

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

The goal of the derivation is to fuse the occurrence of `bottomup` appearing in the definition of the strategy `red` called by `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 definitions in order to arrive at a single expression containing the complete specification of the `innermost` strategy. The `bottomup` strategy can then be distributed over the right-hand sides of the rules to which `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 -> r` is just syntactic sugar for `L = {x1,...,xn:?l;!r}`, where the `xi` are the variables occurring in the rule. The example rules from module `peano` in Figure 1 thus reduce to

```
A = {x: ?Plus(Zero, x); !x}
B = {x,y: ?Plus(Succ(x), y); !Succ(Plus(x, y))}
```

4.2 Inlining Definitions

The first step of the derivation consists in inlining the definitions, i.e., in replacing calls to strategies by their bodies. If `f(s1,...,sn) = s` is the definition of strategy operator `f`, then a call `f(s1,...,sn)` to that operator can be replaced by `s[s1/x1,...,sn/xn]`, 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 `main` strategy in module `apply-peano` in Figure 2, inlining gives

```
(1) innermost(A + B)
```

By the definition of `innermost` this expands to

```
(2) bottomup(red(A + B))
```

By the definition of `red`, this in turn gives

```
(3) bottomup(rec r((A + B); bottomup(r) <+ id))
```

Finally, inlining the definitions of rules `A` and `B` gives

```
(4) bottomup(rec r(
      ( {x:   ?Plus(Zero, x); !x}
      + {x,y: ?Plus(Succ(x), y); !Succ(Plus(x, y))}
      ); bottomup(r)
     <+ id))
```

4.3 Sequential Composition over Choice

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

```
(x + y); z -> (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 `id` always succeeds, `r` in the recursive strategy `(4)` is guaranteed to succeed as well. Thus, `bottomup(r)` is guaranteed to succeed, and so right distribution of `bottomup(r)` according to the rule is valid. This gives

```
(5) bottomup(rec r(
        {x:   ?Plus(Zero, x); !x}; bottomup(r)
      + {x,y: ?Plus(Succ(x), y); !Succ(Plus(x, y))}; bottomup(r)
     <+ id))
```

4.4 Sequential Composition over Scope

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

```
{xs: s1}; s2 -> {xs: s1; s2}
```

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

```
(6) bottomup(rec r(
         {x:   ?Plus(Zero, x); !x; bottomup(r)}
       + {x,y: ?Plus(Succ(x), y); !Succ(Plus(x, y)); bottomup(r)}
      <+ id))
```

4.5 Strategy Application

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

```
(7) bottomup(rec r(
         {x:   ?Plus(Zero, x); <bottomup(r)> x}
       + {x,y: ?Plus(Succ(x), y); <bottomup(r)> Succ(Plus(x, y))}
      <+ id))
```

4.6 Distribution of Bottomup

The application of `bottomup(r)` to a constructor application leads to the following derivation:

```
    <bottomup(r)> c(t1,...,tn)
  = {definition of bottomup}
    <rec x(all(x); r)> c(t1,...,tn)
  = {recursion}
    <all(rec x(all(x); r)); r> c(t1,...,tn)
  = {semantics of sequential composition and all}
    <r> c(<rec x(all(x); r)>t1,..., <rec x(all(x); r)>tn)
  = {definition of bottomup}
    <r> c(<bottomup(r)> t1,...,<bottomup(r)> tn)
```

By repeatedly applying this rule, `bottomup(r)` is distributed over the term constructions in the right-hand sides until variables are encountered. This gives

```
(8) bottomup(rec r(
       {x :  ?Plus(Zero, x); <bottomup(r)> x}
     + {x,y: ?Plus(Succ(x), y);
             <r> Succ(<r> Plus(<bottomup(r)> x,<bottomup(r)> y))}
    <+ id))
```

4.7 Avoiding Renormalization

Finally, we use the observation that

```
    <bottomup(r)> v -> v
```

if `v` is a variable originating in the left-hand side of a rule. That is, if `vs` contains all variables occurring in `l`, `v` is in `vs`, and `{vs:?l;!r}` is a strategy, then occurrences of `<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 `innermost(A+B)`:

```
(9) bottomup(rec r(
          {x:    ?Plus(Zero, x); !x}
        + {x, y: ?Plus(Succ(x), y); <r> Succ(<r> Plus(x, y))}
       <+ 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 `innermost` strategy in Stratego specifications. The optimization works for all strategies of the form `innermost(R1 + ... + Rn)` with the `Ri` rules (and not just for our running example).

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) Scope(["x"],
       Seq(Match(Op("Plus",[Op("Zero",[]),Var("x")])),
           Build(Var("x"))))
```

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

```
(11) Scope(["x","y"],
       Seq(Match(Op("Plus",[Op("Succ",[Var("x")]),Var("y")])),
           Build(Op("Succ",[Op("Plus",[Var("x"),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 `[t1,...,tn]` is also provided.

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.

```
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.

```
module strategy-patterns
overlays
  Do(x) = Call(SVar(x),[])
  Innermost(s, im, r, y) = Bottomup(im, Red(s, r, y))
  Bottomup(r, s) = Rec(r, Seq(All(Do(r)), s))
  Red(s, x, y) = Rec(x, LChoice(Seq(s, Bottomup(y, Do(x))), Id))
```

Fig. 6. Abstract syntax patterns for several standard traversal strategies.

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. 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 3. The "extra" parameters in Figure 6 correspond to bound variables from Figure 3.

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 `BottomupOver-Constructor` distributes `Bottomup` over constructor application. The rule uses the `map` strategy 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`.

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 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,r,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 3. 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 `progagate-mark` to `(13)` then gives

```
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.

```
(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")))))
```

```
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.

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 `s` to each child. The argument to `alltd` in the rule `BottomupToVarIsId` is the strategy derived from the local rule

```
App(Bottomup(_,r), Var(v)) -> Var(v)
where <fetch(?v)> vs; <isr> r
```

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

Applying the `alltd` traversal specified in `fusion` to `(15)`, for example, gives

```
(18) Bottomup(p,Rec(w,LChoice(Choice(alpha2,beta2),Id)))
```

where `alpha2` is

```
(19) Scope(["x"],
       Seq(Match(Op("Plus",[Op("Zero",[]),Var("x")])),
           Build(Var("x"))))
```

and `beta2` is

```
(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")]))])))))
```

This is the final result of applying `fusion` to the abstract syntax representation of `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 optimization phase in the stratego compiler. The optimization still has some limitations. Only strategies of the form `innermost(R1 + ... + Rn)`, where the `Ri` are strategies are optimized. Strategies such as `innermost(rules1 + rules2)`, where `rules1` and `rules2` are defined as `rules1 = R1_1 + ... + R1_k` and `rules2 = R2_1 + ... + R2_l` are not handled properly because the inliner currently does not inline such definitions. This requires a generalization of the inliner. Furthermore, rules with conditions that introduce new variables are handled, but the terms bound to the newly introduced variables are renormalized. Based on an analysis of the conditions this could be avoided under some circumstances.

Strategies have also been used to optimize programs which are not themselves defined in terms of strategies. In [7], for example, they are used to eliminate intermediate data structures from functional programs. In [16], strategies are used to build optimizers for an intermediate format for ML-like programs. In both cases, strategies are used — as they are here — in conjunction with small local transformations to achieve large-scale optimization effects.

Small local transformations have been dubbed “humble transformations” in [13]. Such transformations are used extensively in optimizing compilers based on the compilation-by-transformation idiom [8, 9, 1, 14]. They are also used to some degree in most others compilers.

The optimization of `innermost` presented in this paper was inspired by more general work on functional program optimization. In [6], an optimization scheme for compositions of functions that uniformly consume algebraic data structures with functions that uniformly produce substitution instances of them is given. This scheme is generic over data structures, and has been proved correct with respect to the operational semantics of Haskell-like languages. One aspect of future work will involve more completely incorporating the ideas underlying this scheme into strategy languages to arrive at more generally applicable and provably correct optimizations of strategy-based program patterns. In particular, we aim to see the `innermost` fusion technique described in this paper as the specialization to `innermost` of a generic, automatable, and provably correct fusion strategy which is provably correct with respect to the semantics in [16].

The importance of optimizing term traversals in functional transformation systems is discussed in [5]. Term traversals are modelled there by fold functions but, since the fold algebras under consideration are updatable, standard fusion techniques for functional programs [17, 10, 18] are not immediately applicable. The fusion techniques presented here may nevertheless provide a means of implementing optimizations which automatically shortcut recursion in term traversals. If, as suggested in [5], shortcuts of recursion in term traversals should be regarded as program specialization then, since specialization can be seen as an automated instance of the traditional fold/unfold program optimization methodology [12], optimization of traversals should indeed be achievable via fold/unfold transformations. These connections are deserving of further investigation.

Finally, measurements to express the optimizations achieved by `innermost` fusion and related fusion techniques are needed.

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

References

1. A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
2. T. Dinesh, M. Haveraaen, and J. Heering. An algebraic programming style for numerical software and its optimization. *Scientific Programming*.
3. P. Fradet and D. L. Métayer. Compilation of functional languages by program transformation. *ACM Trans. Prog. Lang. Syst.*, 13(1):21–51, Jan. 1991.
4. M. Haveraaen, H. A. Friis, and T. A. Johansen. Formal software engineering for computational modeling. *Nordic Journal of Computing*, 6(3):241–270, 1999.
5. R. L. J. Kort and J. Visser. Functional transformation systems. http://www.cwi.nl/ jvisser/papers/, September 2000. Presented at 9th International Workshop on Functional and Logic Programming (WFLP'2000).
6. P. Johann. Short cut fusion: Proved and improved. Technical report, Bates College, 2000.
7. P. Johann and E. Visser. Warm fusion in Stratego: A case study in the generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):1–34, 2000.
8. R. Kelsey and P. Hudak. Realistic compilation by program transformation. In *ACM Conference on Principles of Programming Languages*, pages 281–292, January 1989.
9. R. A. Kelsey. *Compilation by Program Transformation*. PhD thesis, Yale University, May 1989.
10. J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In S. L. P. Jones, editor, *Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 314–323. ACM Press, June 1995.
11. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
12. C. G. N. Jones and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
13. S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, September 1998.
14. K. H. W. P. S. L. Peyton Jones, C. Hall and P. Wadler. The glasgow haskell compiler: a techical overview. In *UK Joint Framework for Information Technology (JFIT) Technical Conference*, March 1993.

15. E. 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)*, Lecture Notes in Computer Science. Springer-Verlag, May 2001. To appear.
16. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).
17. P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.
18. H. I. Y. Onoue, Z. Hu and M. Takeichi. A calculational fusion system HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106. Chapman and Hall, February 1997.