Strategies for Fusing Logic and Control via Local, Application-Specific Transformations

Patricia Johann
Eelco Visser

Technical Report UU-CS-2003-050
Institute of Information and Computing Sciences
Utrecht University

February 2003

ISSN 0924-3275

Address:
Patricia Johann
Department of Computer Science, Rutgers University
Camden, NJ 08102, U.S.A.,
`http://www.crab.rutgers.edu/~pjohann`,
`pjohann@crab.rutgers.edu`

Eelco Visser
Institute of Information and Computing Sciences
Utrecht University
P.O.Box 80089
3508 TB Utrecht
`http://www.cs.uu.nl/~visser`
`visser@acm.org`

Strategies for Fusing Logic and Control via Local, Application-Specific Transformations

Patricia Johann[1] and Eelco Visser[2]

[1]*Department of Computer Science, Rutgers University, Camden, NJ 08102, U.S.A.,* `http://www.crab.rutgers.edu/~pjohann`, *pjohann@crab.rutgers.edu*

[2]*Institute of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands,* `http://www.cs.uu.nl/~visser`, *visser@acm.org*

Abstract

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.

Stratego is a language for program transformation based on the paradigm of rewriting strategies. In Stratego, transformation rules define basic transformation steps and user-definable strategies control the application of rules to a program. Since the problem-specific rules and the highly generic strategies which apply them are kept separate, these elements can be combined in a mix-and-match fashion to produce a variety of program transformations. In some instances this separation of concerns leads to inefficient implementations.

In this paper we show how such inefficiencies can be remedied using fusion. Furthermore, we show how fusion can be implemented using rewriting strategies by studying in detail the application of rewriting strategies to the fusion of the generic innermost strategy with sets of arbitrary-but-specific rewrite rules. Both the optimization and the programs to which the optimization applies are specified in Stratego.

The contributions of this work are twofold. In the first place, we show how to optimize and reason about rewriting strategies, which opens up a new area of strategy optimization. In the second place, we demonstrate how such optimizations can be implemented effectively using local, application-specific transformations. These techniques are applicable to transformation of programs in languages other than Stratego.

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 composing appropriate specializations of generic components. Generic components can be specialized in many different ways, and their resulting instances can be reused in many combinations. The advantages of abstract programming are reduced code size and increased modularity of programs.

One disadvantage of abstract programming is that the separation it supports between logical concerns and issues of control in program construction can introduce considerable overhead, even for simple computations. By contrast, code written specifically to implement one particular problem instance can effectively intermingle logic and control to arrive at a more efficient implementation than is possible generically. The challenge of abstract programming is to maintain a high-level separation of concerns while simultaneously achieving the efficiency of such intermingled programs.

1.1. Fusing Compositions of Abstract Programs

Fusion techniques mitigate the tension between modularity and efficiency by automatically deriving more efficient versions of programs from their abstract composite versions. This is achieved by intermingling pieces of generic components with code relevant to specific problem instances. In deforestation of functional programs, for example, intermediate data structures are eliminated by fusing together function compositions [23, 28]. In the Sophus style of numeric programming, fusion enables transformation from an algebraic style of programming resembling the mathematical specification of numeric programs to an updating style in which function arguments are overwritten in order to reuse memory allocated to large matrices [4, 8].

Since fusion can optimize both individual software components and complete applications constructed from them, its automation packs enormous potential for programming-in-the-large. While an experienced programmer might easily optimize a small abstract program at the keyboard, hand optimization of even modest programs can be both difficult and error-prone. And as programs become larger and more complex, the difficulty of fusing programs by hand becomes even more pronounced. Automatic fusion tools allow the programmer to program in a more abstract style without compromising program efficiency.

1.2. Abstract Programming in Stratego

In this paper we consider abstract programming in, and fusion techniques for, Stratego [25, 27]. Stratego is a domain-specific language for the specification of program transformation systems based on the paradigm of rewriting strategies. It separates the specification of basic transformation rules from the specification of strategies by means of which those rules 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, which in turn allows strategies and rules to be composed as necessary to achieve various program transformations. The resulting abstract style of programming 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.

1.3. Generic Innermost Strategy

One pattern of control of particular interest is innermost term traversal. Innermost traversal has been the foundation of successful evaluation mechanisms for a variety of programming languages, and has also found widespread use in automated deduction. An innermost evaluation strategy can be obtained by specializing a generic innermost traversal strategy — parameterized by a particular set of rules to be applied to the subject term during an innermost traversal — to a set of evaluation rules; innermost normalization strategies suitable for other applications can be obtained in a similar manner. Despite the ease with which generic innermost traversal strategies can be defined in Stratego, efficiency concerns prevented most Stratego programmers from using them much in practice.

The innermost normalization strategy `innermost(s)` takes as argument a transformation `s` and normalizes terms with respect to this transformation. An application of `innermost` of the form `innermost(R1 <+ ... <+ Rn)` can be optimized for the specific transformation rules `Ri`. Innermost normalization proceeds by traversing a subject term from the bottom up, applying the specified transformation rules to each subterm. However, whenever a rule `Ri` successfully reduces a subterm, the strategy `innermost(R1 <+ ... <+ Rn)` must be applied bottom-up to the reduct as well in order to normalize it. As a result, subterms of the reduct which are also subterms of the original subterm are reconsidered for normalization. In fact, depending on the structure of the right-hand side of `Ri`, subterms of the reduct that correspond to variables on the left-hand side of `Ri` may be reconsidered for normalization a number of times. Of course, these subterms were already normalized before `Ri` was applied to the subject term, and so they need not be considered for renormalization at all! Fusing the control captured by `innermost` with the logic embodied in a set of rules `R1`, ..., `Rn` yields an implementation of `innermost(R1 <+ ... <+ Rn)` in which renormalization of such “variable subterms” is completely avoided.

1.4. Fusing Innermost with Transformation Rules

In this paper we develop a transformation on Stratego programs which fuses the generic innermost traversal with the set of rules with which it is instantiated. This optimization is fully automatable, and is implemented in Stratego itself. In

Section 8 we prove it correct with respect to the operational semantics of the Stratego programs it transforms.

The contributions of this work are twofold. In the first place, we show how reasoning about and optimizing strategies based on rewriting can proceed, opening up a new area of symbolic computation with strategies. In the second place, we demonstrate the effective implementation of such optimizations in Stratego using a combination of interesting techniques and idioms. Writing transformations in the *concrete syntax* of the object language makes the specification much easier to understand [26]. The use of *dynamic rules* allows the definition of context-sensitive transformations [24]. Finally, we introduce the idiom of *cascading*, *local*, and *application-specific transformations*, which allows the specification of complex transformations using sets of simple rules.

The application of the techniques in this paper is not restricted to the particular optimization of innermost traversals studied here, but can also be used for other optimizations on Stratego programs, as well as for transformations on programs in other languages. The fusion optimization described in this paper and all techniques used to implement it are part of the Stratego compiler (version 0.9[*]).

1.5. Outline

The remainder of this paper is organized as follows. In Section 2 we cover the basics of Stratego and introduce the generic Stratego specification of innermost normalization. In Section 3 we explore different idioms for program transformation and sketch their realization in Stratego. In particular, we explain the concepts of local and application-specific transformations. In Section 4 we discuss some shortcomings of the modular specification of `innermost`, present an optimized version of the innermost strategy, and argue that automatic derivation of optimized programs from modular ones is needed. Section 5 shows, informally, how the optimized specification of innermost can be derived in a systematic way from the original specification. In Sections 6 through 8 we discuss the implementation of this transformation in Stratego. Section 6 formalizes the syntax and semantics of Stratego. Section 7 explains how Stratego programs can be used to transform other Stratego programs. Section 7.4 formalizes the transformation rules from Section 5. Section 8 presents the strategy that combines these rules into the complete fusion transformation. Novel uses of Stratego are highlighted as they are used in defining the fusion transformation strategy, and the correctness of each transformation is established as it is introduced. Finally, Section 9 discusses previous, related, and future work.

[*]See `http://www.stratego-language.org` for the distribution of the Stratego compiler and library.

2. Rewriting Strategies in Stratego

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 programmable rewriting strategies control the application of those rules.

In this section we informally describe the subset of Stratego that is the object of transformations in this paper. To illustrate it, we show a small specification which simplifies expressions over natural numbers with addition using a generic specification of innermost normalization. A formal description of the syntax and operational semantics of the subset of Stratego with which we are concerned in this paper is given in Section 6.

2.1. Signatures and Terms

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. Note that terms are variable-free by definition.

2.2. 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`. We write `Vars(p)` for the variables occurring in the pattern `p`, and regard terms as term patterns containing no variables. Figure 1 shows rewrite rules `A` and `B` that simplify sums of natural numbers. As indicated there, Stratego provides a simple module structure that allows modules to import other modules.

```
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))
```

Figure 1: An example Stratego module with signature and rewrite rules.

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

Figure 2: Using Peano rules.

A rule `L: l -> r` applies to a term `t` when the pattern `l` matches `t`, i.e., when the variables of `l` can be replaced by terms in such a way that the result is precisely `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 the term `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 it 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 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.

2.3. 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`. This strategy can be instantiated with any selection of rules to achieve normalization of terms under those rules. In Figure 2, for instance, the strategy `main` is defined to normalize `Nat` terms using the innermost strategy instantiated with rules `A` and `B` from Figure 1. 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. It is thus possible in Stratego to develop a library of valid transformation rules to be applied under various strategies as needed.

2.4. 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 (which can, of course, be the original term). In the case of failure, there is no result.

```
module innermost
strategies
  try(s)       = s <+ id
  bottomup(s)  = all(bottomup(s)); s
  innermost(s) = bottomup(try(s; innermost(s)))
```

Figure 3: Generic traversal strategies.

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 *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)`. The *negation* `not(s)` of a strategy `s` succeeds with the identity transformation if `s` fails, and fails if `s` succeeds.

A strategy definition `f(x1,...,xn) = s` introduces a new strategy operator `f` parameterized with strategies `x1,...,xn` and having body `s`. Definitions can be recursive. In fact, the recursion operator `rec x(s)` is shorthand for a recursive definition `let x = s in x`.

2.5. 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 to a subterm of a subject term, the term must be traversed.

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. For each constructor `C` there is a corresponding congruence operator, also denoted `C`. If `C` is an `n`-ary constructor, then the corresponding congruence operator defines the strategy `C(s1,...,sn)`. Such a strategy applies only to terms of the form `C(t1,...,tn)`. It results in the term `C(t1',...,tn')`, provided the application of each strategy `si` to each term `ti` succeeds with result `ti'`. If the application of `si` to `ti` fails for any `i`, then the application of `C(s1,...,sn)` to `C(t1,...,tn)` also fails.

While congruence operators support the definition of traversals that are *specific* to a data type, Stratego also provides combinators for composing *generic traversals*. The operator `all(s)` applies `s` to each of the direct subterms `ti` of a constructor application `C(t1,...,tn)`. It succeeds if and only if the application of `s` to each direct subterm succeeds. In this case the resulting term is the constructor application `C(t1',...,tn')`, where each term `ti'` is obtained by applying `s` to `ti`. Note that `all(s)` is the identity on constants, i.e., on constructor applications without children.

An example of the use of `all` is the definition of the strategy `bottomup(s)` in Figure 3. There, the strategy expression `(all(bottomup(s)); s)` specifies that `s` is first applied recursively to all direct subterms — and thus to all subterms — of the subject term. If that succeeds, then `s` is applied to the resulting term. This definition of `bottomup` thus captures the generic notion of a bottom-up traversal over a term.

2.6. Innermost Normalization

The `innermost` strategy in Figure 3 is defined using `bottomup`. It performs a bottom-up traversal over a term, applying the strategy `try(s;innermost(s))` to each subterm. Thus, for each subterm, after applying `try(s;innermost(s))` to each of its subterms, the transformation `s` is applied to it. If that succeeds, the reduct resulting from the transformation is recursively normalized. If `s` fails, however, the subterm must be in `s`-normal form.

The `innermost` strategy in Figure 3 captures the notion of parallel innermost reduction. Other specifications of innermost normalization are possible since Stratego representations of strategies are not, in general, unique. In particular, by unfolding the definition of `innermost` twice and then folding again using a local recursion, we arrive at the following alternative specification:

```
  innermost(s)
= bottomup(try(s; innermost(s)))
= bottomup(try(s; bottomup(try(s; innermost(s)))))
= bottomup(rec r(try(s; bottomup(r))))
```

3. Transformation Techniques and Idioms

In this section we will briefly outline the extension of Stratego with *concrete syntax* and *dynamic rules*, and then discuss a number of useful transformation idioms and show how Stratego naturally supports their implementation. On the one hand, these idioms *motivate* the use of programmable strategies. On the other, they are *tools* that will be applied in the fusion transformation later in this paper.

3.1. Concrete Syntax

In the previous section we saw that Stratego can be used to define transformations on abstract syntax trees representing the programs to be transformed, rather than on their text-based representations. But the direct manipulation of abstract syntax trees can be unwieldy for larger program fragments. Therefore, Stratego supports the specification of transformation rules using the *concrete syntax* of the object language [26]. For example, using concrete syntax, one can declare the following transformation on strategy expressions:

```
SeqOverLChoiceR :
  |[ (s1 <+ s2); s3 ]| -> |[ (s1; s3) <+ (s2; s3) ]|
```

instead of the equivalent transformation on abstract syntax trees:

```
SeqOverLChoiceR :
  Seq(LChoice(s1, s2), s3) -> LChoice(Seq(s1, s3), Seq(s2, s3))
```

Concrete syntax is merely syntactic sugar for the specification of transformations on its corresponding abstract syntax.

3.2. Dynamic Rules

Programmable rewriting strategies provide control over the application of rewrite rules. But one limitation of pure rewriting is that rewrite rules are context-free. That is, a rewrite rule can only use information obtained by pattern matching on the subject term or, in the case of conditional rewriting, from the subterms of the subject term. Yet, for many transformations, information from the context of a program fragment is needed. The extension of strategies with *scoped dynamic rules* [24] makes it possible to access this information.

Unlike standard rewrite rules in Stratego, dynamic rules are generated at run-time, and can access information available from their generation contexts. For example, in the following strategy, the transformation rule `InlineStrat` defines the replacement of a call `f(ss)` by the appropriate instantiation of the body `s` of its definition:

```
declare-inline-rule =
  ?|[ f(as) = s ]|;
  rules(
    InlineStrat :
      Strategy|[ f(ss) ]| -> <ssubs> (as', ss, s')
      where <strename> |[ f(as) = s ]| => |[ f(as') = s' ]|
  )
```

The rule `InlineStrat` is generated in the context of the *definition* of the strategy `f`, but applied at the *call sites* `f(ss)`. (See Section 8 for more details.) Dynamic rules are first-class entities and can be applied as part of a global term traversal. It is possible to restrict the application of dynamic rules to certain parts of subject terms using rule scopes, which limit the live range of rules.

3.3. Cascading Transformations

The basic idiom of program transformation achieved with term rewriting is that of *cascading transformations*. Instead of applying a single complex transformation algorithm to a program, a number of small, independent transformations are applied in combination throughout a program or program unit to achieve the desired effect. Although each individual transformation step achieves little, the cumulative effect can be significant, since each transformation feeds on the results of the ones that came before it. Cascading transformations are the key, for example, to the compilation-by-transformation approach [5] applied in the Glasgow Haskell Compiler [20]. GHC applies a large number of small, almost trivial program transformations throughout programs to achieve large-scale optimization by accumulating small program changes.

One common cascading of transformations is accomplished by exhaustively applying rewrite rules to a subject term. In Stratego the definition of a cascading normalization strategy with respect to rules `R1`, ... ,`Rn` can be formalized using an `innermost` strategy:

```
simplify = innermost(R1 <+ ... <+ Rn)
```

However, other strategies are possible. For example, the GHC simplifier applies rules in a single traversal over a program tree in which rules are applied both on the way down and on the way up. This is expressed in Stratego by the strategy

```
simplify  = downup(repeat(R1 <+ ... <+ Rn))
downup(s) = s; all(downup(s)); s
repeat(s) = try(s; repeat(s))
```

3.4. Staged Transformations

Cascading transformations apply a number of rules one after another to an entire program. But in some cases this is not appropriate. For instance, two transformations may be inverses of one another, so that repeatedly applying one and then the other would lead to non-termination. To remedy this difficulty, Stratego supports the idiom of *staged transformation*.

In staged computation, transformations are not applied to a subject term all at once, but rather in stages. In each stage, only rules from some particular subset of the entire set of available rules are applied. In the TAMPR program transformation system [2, 3] this idiom is called *sequence of normal forms*, since a program tree is transformed in a sequence of steps, each of which performs a normalization with respect to a specified set of rules. In Stratego this idiom can be expressed directly as

```
simplify =
  innermost(A1 <+ ... <+ Ak)
  ; innermost(B1 <+ ... <+ Bl)
  ; ...
  ; innermost(C1 <+ ... <+ Cm)
```

Staged transformations can be applied fruitfully in combination with cascading transformations: a transformation is expresssed as a sequence of stages, where each stage is a cascading transformation. Indeed, the GHC simplifier mentioned above, is in effect a staged transformation in which the simplification traversal over a program tree is repeated and alternated with various analyses. On the other hand, the steps in a staged transformation can use quite different idioms from one another, and can even involve complex monolithic computations.

The advantage of separating rules from strategies is particularly compelling in this case of staged transformations. Since rules are defined independently of the particular stages in which they are used, it is easy to reuse them in many different stages.

3.5. 'Local' Transformations

In conventional program optimization, transformations are applied throughout a program. In optimizing imperative programs, for example, complex transformations are applied to entire programs [17]. In GHC-style compilation-by-transformation, small transformation steps are applied throughout programs.

In this paper we introduce a style of transformation that is a mixture of these ideas. Instead of applying a complex transformation algorithm to a program we use staged, cascading transformations to accumulate small transformation steps for large effect. However, instead of applying transformations throughout the subject program, we often wish to apply them locally, i.e., only to selected parts of the subject program. This allows us to use transformations rules that would not be beneficial if applied everywhere.

One example of a strategy which achieves such a transformation is

```
transformation =
  alltd(
    trigger-transformation
    ; innermost(A1 <+ ... <+ An)
  )
alltd(s) = s <+ all(alltd(s))
```

The strategy `alltd(s)` descends into a term until a subterm is encountered for which the transformation `s` succeeds. In this case the strategy `trigger-transformation` recognizes a program fragment that should be transformed. Thus, cascading transformations are applied locally to terms for which the transformation is triggered. Of course more sophisticated strategies can be used for finding application locations, as well as for applying the rules locally. Nevertheless, the key observation underlying this idiom remains: Because the transformations to be applied 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.

3.6. Application-Specific Transformations

Optimizers are usually based on *generic transformations*, i.e., on transformations derived from the semantics of the programming language under consideration, and generic analyses applied to its subject programs. However, when knowledge about the specific application at hand is available, it is sometimes possible to achieve better results. *Application-specific transformations* allow special knowledge about a specific application, or about a specific library to be used in transformations.

Conventional fusion techniques for functional programming languages — such as `foldr/build` fusion [6], `destroy/unfoldr` fusion [22], and hylo fusion [29] — are based on application-specific transformations. The `foldr/build` rule, for example, relies on programs being written in terms of the specialized program constructs `build` and `foldr`, which encode uniform production and consumption of algebraic data structures, respectively. The rule assumes that these constructs exhibit particular operational behaviors — although this is in no way verified by the compiler before fusion — and performs a program transformation which is correct with respect to that assumption. It is now possible to provide such application-specific transformations to GHC via user-definable rewrite rules which are applied as part of simplification [18].

The fusion transformation on `innermost` in this paper also relies on library knowledge rather than on general program analysis. In order to ensure that the `innermost` strategy used in a program conforms to its expected semantics, its definition is matched against our own library definition. Only when matching is successful is the strategy inlined and the call then optimized. The general approach of the transformation is based on the cascading style in which locally applicable transformation rules are applied. However, the key step in the transformation is based on insight into the algorithm, rather than syntactic manipulation, and our transformation can be considered an encoding of this insight. Thus, using syntax-directed application-specific program transformations we can achieve a greater degree of optimization more effectively than is possible with only general program analysis.

4. An Optimized Specification of Innermost Reduction

In the previous sections we have seen that a variety of generic rewriting strategies can be composed from a few combinators, and that these strategies can in turn be used to encode a wide variety of transformation idioms. Moreover, this can all be done even while preserving the separation between rules and strategies. Such generality does, however, come at a price. 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 again the transformation of a term with the `innermost(A <+ B)`

```
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
    ))
```

Figure 4: Optimized strategy.

strategy from Figure 2. By the definition of `innermost`, when `(A <+ B)` is applied to a subterm, all its proper subterms are already in normal form. For example, in any term matching the left-hand side `Plus(Succ(x),y)` of the rule `B`, the subterms corresponding to the variables `x` and `y` must already have been normalized. Nevertheless, once the appropriate instance of the right-hand side `Succ(Plus(x,y))` of `B` is constructed, it must be normalized with `innermost(A <+ B)`. This entails that the terms bound to `x` and `y` are completely renormalized: These "variable subterms" are completely traversed, and the rules `A` and `B` are tried at each of their subterms. Since the variable subterms terms are in normal form, no actual transformation occurs, of course, but the specification of `innermost` in Figure 3 still requires their traversal. Traversing in its entirety each subterm resulting from each application of each rule leads to suboptimal performance of the normalization strategy.

4.1. Optimizing Innermost

The suboptimal complexity of the `innermost` strategy is a direct result of the separation of concerns between the generic strategy and the rules with which it is parameterized. With knowledge of the specific rules to be used in normalization in hand, an efficient implementation of the `innermost` strategy can be achieved.

For example, in the specific case of innermost normalization with respect to the rules `A` and `B`, an efficient definition is the one given in Figure 4. This definition completely avoids renormalization of variable subterms. Like the original definition of `innermost(A <+ B)`, the optimized definition performs innermost normalization of a term with respect to rules `A` and `B`. However, rather than renormalizing *every* subterm of the term resulting from a single-step reduction by `A` or `B`, the optimized strategy recursively applies the reduction strategy only to the non-variable subterms of the reduct.

Examination of the optimized strategy shows how this is accomplished. Like the original `innermost` strategy, the strategy performs a bottom-up traversal. At each node, a recursive strategy `r` tries to apply either rule `A` or rule `B`, or else vacuously succeeds with `id`. Instead of separating the rules from the strategy, the rules have been intertwined with it. To do this most effectively, the rules have been rephrased in terms of the true primitive actions of rewriting, i.e., pattern matching (`?pat`) and pattern instantiation (`!build`). Thus, rule `A` is rephrased as

```
{x: ?Plus(Zero, x); !x}
```

and rule `B` is rephrased as

```
{x,y: ?Plus(Succ(x), y); !<r> Succ(<r> Plus(x,y))}
```

In these rephrasings, the scope of the pattern variables `x` and `y` is delimited by the scope construct `{x1,...,xn:s}`. (See Section 6 for formal definitions of `?pat`, `!pat`, and variable scope.)

Now, instead of applying the complete `innermost` strategy to the reduct of a rule application, the inner recursive strategy `r` is invoked *only* on those subterms of the reduct that may not be in normal form. In particular, `r` is not applied to variable subterms from the left-hand side of the rule, since these are already known to be in normal form. The notation `<s> t` denotes the application of strategy `s` to the instance of the term `t` determined by the current bindings. The first scoped strategy above thus replaces a subject term of the form `Plus(Zero, t)` with its subterm `t`. Similarly, the second replaces a term of the form `Plus(Succ(t), t')` with the term obtained by applying the strategy `r` to the term obtained by applying `Succ` to the term obtained by applying `r` to `Plus` applied to `t` and `t'`.

4.2. Effect of the Optimization

The entangling of rules and strategy is quite effective, as can be seen from the benchmark results in Figure 5. In the benchmark, `innermost(A <+ B)` is used to reduce terms of the form `Plus(Succ(...), Zero)` nested n deep. Since the reduction of each `Plus` requires $n + 1$ steps, the entire reduction requires $\frac{(n+3)n}{2}$ steps. The results show that the run-time for the original strategy grows much faster than that of the optimized version.

By fusing strategies and the rules they apply, significantly more efficient implementations of transformations can be achieved. However, writing such tangled specifications of rewrite systems is not attractive since it leads to much more complex specifications, as well as to specifications in which rules are tied to specific transformations and so cannot be reused in other ones. Furthermore, the explicit recursive invocation of the transformation in the right-hand side of rules make it less clear what belongs to the original transformation rule, and what belongs to the strategy. This makes understanding, and therefore maintaining, the rules more difficult.

Thus, while the entangled specification is more efficient, the modular one is better for development. Automatically fusing the generic strategy with its arguments is therefore attractive. As we will see, it is also possible. In the next section we will show that the optimized version can be derived from the generic version. In Section 8 we will show that automatic fusion can derive the optimized

n	#rewrites	plain		fused	
		time	rewr/sec	time	rewr/sec
25	350	0.01	35000.00	0.00	-
30	495	0.02	24750.00	0.00	-
35	665	0.02	33250.00	0.00	-
40	860	0.03	28666.67	0.00	-
45	1080	0.05	21600.00	0.00	-
50	1325	0.07	18928.57	0.00	-
100	5150	0.47	10957.45	0.01	515000.00
200	20300	3.55	5718.31	0.01	2030000.00
300	45450	11.88	3825.76	0.02	2272500.00
400	80600	27.93	2885.79	0.05	1612000.00
500	125750	55.66	2259.25	0.06	2095833.33
600	180900	95.05	1903.21	0.09	2010000.00
700	246050	144.17	1706.67	0.11	2236818.18
800	321200	196.38	1635.60	0.14	2294285.71
900	406350	280.07	1450.89	0.20	2031750.00
1000	501500	384.47	1304.39	0.24	2089583.33
2000	2003000	3115.09	643.00	1.43	1400699.30

$$\begin{aligned} f(0) &= \texttt{Zero} \\ f(n+1) &= \texttt{Plus(Succ(}f(n)\texttt{),Zero)} \end{aligned}$$

Figure 5: Benchmark results for unoptimized and optimized transformation `innermost(A <+ B)` applied to terms generated by $f(n)$. The table shows the number of rewrites, i.e., rule applications that are performed, and for each of the strategies, the time in seconds, and the number of rewrite steps per second. The benchmarks were performed on a 2GHz Pentium 4 with 1GB RAM (of which the benchmark uses only 0.5%.)

definition of `innermost` from its generic version. That the original and optimized versions have the same operational semantics will be proved in that section.

5. Deriving the Optimized Specification

In this section we show how the optimized implementation of Figure 4 can be derived from the strategy `innermost(A <+ B)` by systematic transformation. Although we demonstrate the derivation technique by applying it to the specific program `innermost(A <+ B)` from Figure 2, it can be used to optimize the application of `innermost` to any selection of rules equally well. In the remainder of the paper we will formalize in Stratego the transformation rules we use here, and then develop a strategy for automatically applying them in the correct order.

The goal of the derivation is to fuse the recursive invocation of `innermost` with the right-hand sides of the rules `A` and `B`. To achieve this, we first desugar the rules and then inline (unfold) strategy 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` is applied.

5.1. Desugaring Rules

In Stratego, rules are not primitives. Instead, rules are expressed in terms of the constructs `?t` and `!t`. A rule `L: l -> r` is just syntactic sugar for the strategy definition `L = {x1,...,xn:?l;!r}`, where the `xi` are the variables occurring in the rule. The similarities between the original definition of `innermost` from Figure 2 and the optimized definition from Figure 4 become more apparent once we observe that the rules `A` and `B` from Figure 1 represent the strategy definitions

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

and

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

respectively.

5.2. Inlining Definitions

The first step of the derivation consists in *inlining* definitions, i.e., in replacing each call to a strategy by the body of its definition. If `f(x1,...,xn) = 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 Figure 2, inlining gives

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

By the definition of `innermost` and the derivation on page 10 this expands to

```
(2) bottomup(rec r(try((A <+ B); bottomup(r))))
```

Inlining the definitions of rules `A` and `B` gives

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

5.3. Sequential Composition over Choice

In the next step of the derivation we right distribute the bottomup strategy over the deterministic choice operator 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. The strategy `(x <+ y); z` will fail if application of `x` is attempted, but `(x;z) <+ (y;z)` will always succeed since `(y;z)` does. On the other hand, 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, the strategy `try(s)` succeeds for any parameter strategy `s`. Thus, `r` in the recursive strategy `(3)` is guaranteed to succeed as well. This in turn implies that `bottomup(r)` is guaranteed to succeed, and so right distribution of `bottomup(r)` according to the rule above is valid. This gives

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

5.4. Sequential Composition over Scope

In order to apply `bottomup(r)` to the right-hand sides of the argument rules to `try`, 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 whenever the variables in `xs` are not free in `s2`. Its application transforms `(4)` into

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

5.5. Strategy Application

We can now apply `bottomup(r)` to the term built in the right-hand side of each rule. We get

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

5.6. 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}
    <all(bottomup(r)); r> C(t1,...,tn)
  = {semantics of sequential composition}
    <r> (<all(bottomup(r))> C(t1,...,tn))
  = {semantics of all}
    <r> C(<bottomup(r)>t1,..., <bottomup(r)>tn)
```

By repeatedly applying the rule

```
  <bottomup(r)> C(t1,...,tn) ->
            <r> C(<bottomup(r)> t1,...,<bottomup(r)> tn)
```

`bottomup(r)` can be distributed over the term constructions in the right-hand sides of rules until variables are encountered. Doing so for our running example gives

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

5.7. Avoiding Renormalization

We may now use the observation that

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

if `v` is a variable originating in the left-hand side of a rule to rewrite the right-hand sides of the rule arguments to `try`. In other words, 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 needs, for example, to know which variables in `r'` also appear in `l'`), it does give rise to a transformation that is local in the sense that it is applied only within a single strategy. More specifically, the transformation `<bottomup(r)> v -> v` is applied to a selected part of a program under the control of a particular program transformation strategy. Using it, we arrive at the desired optimized version of `innermost(A <+ B)`:

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

```
sp   := d1 ... dn

d    := f(x1,...,xn) = s

s    := ?mpat | !bpat | {x1,...,xn: s} | where(s) | all(s) |
        s1 ; s2 | s1 <+ s2 | not(s) | f(s1,...,sn) |
        C(s1,...,sn) | rec x(s) | id | fail

mpat := x | C(mpat1,...,mpatn)

bpat := x | C(bpat1,...,bpatn) | <s> bpat

t    := C(t1,...,tn)

p    := (t, E)

E    := {x1 := t1, ..., xn := tn}
```

Figure 6: Abstract syntax of Stratego (subset)

6. Syntax and Semantics of Stratego

In this section we formalize the syntax and operational semantics of Stratego described informally in Sections 2 and 4. Concrete syntax will aid definition of the transformations we use to optimize applications of `innermost`, and a well-defined operational semantics will enable us to prove that the transformations are correct. We formalize only those elements of Stratego that are needed to present our results.

6.1. Syntax

The abstract syntax for the subset of Stratego with which we are concerned in this paper appears in Figure 6. There, `x` is a name that stands for a strategy, term, or pattern, depending on its context. Each numbered item in the figure is of the same syntactic class as its unnumbered counterpart.

A Stratego program `sp` is a listing of strategy definitions. A strategy definition `d` introduces a parameterized strategy operator. A strategy is either an application of the match primitive `?mpat` to a match pattern, an application of the build primitive `!bpat` to a build pattern, a scoped strategy, a `where` clause, an application of the `all` traversal operator to a strategy, a sequential composition of two strategies, a deterministic choice of two strategies, the negation of a strategy, an application of a defined strategy operator to an appropriate number of argument strategies, an application of a congruence operator to an appropriate number of strategies, a recursive closure of a strategy, the identity strategy, or the failure strategy. Most of these constructs have already been introduced informally in Sections 2.4 and 4.1. A formal semantics is given in the next subsection.

A match pattern mpat is either a variable or a fixed-arity constructor applied to an appropriate number of match patterns. A build pattern bpat is either a variable, a fixed-arity constructor applied to an appropriate number of build patterns, or a strategy applied to a build pattern.

Stratego rules can have conditions which are introduced using the keyword where. The operation where(s) applies the strategy s to the subject term. If s succeeds, then the original subject term is restored, and only the newly obtained variable bindings are kept. A conditional rule has the form L: l -> r where(s), and denotes a strategy definition L = {x1,...,xn: ?l; where(s); !r}. The body of the strategy first matches l and then attempts to satisfy the condition s. If s succeeds, then the appropriate instance of r is built. Conditional rules thus apply only if the conditions in their where clauses succeed.

A term t is a fixed-arity constructor applied to an appropriate number of terms. A pair p has a term as its first component and an environment as its second. An environment E is a function from variables to terms with a finite domain. We write dom(E) for the domain of the environment E.

6.2. Operational Semantics

Figure 7 gives the operational semantics of the subset of Stratego with which we are concerned in this paper. The rules in this figure formalize our earlier, informal notion of applying strategies to terms. Formally, strategies are always applied to term-environment pairs to produce new such pairs. If s is a strategy and p is a pair, then we write s @ p to denote the application of s to p. Informally, however, we speak of applying a strategy to a term, with no explicit reference to its accompanying environment.

The symbol F denotes failure of a strategy application. Figure 7 only lists the positive rules for Stratego. But if, for any strategy application s @ p, no rule applies at the root of the term component of p, then the application fails. It is, of course, also possible for computations to be nonterminating.

The notion of one environment extending another is used in the rule for match patterns. We say that E' *extends* E if every binding in E also appears in E'. In this case we write E' > E. In addition, if E is an environment and pat is a pattern, then E(pat) is the instance of pat obtained by replacing all occurrences of variables in pat by their bindings in E. Finally, if E is an environment containing bindings for x1,...,xn, then E-{x1,...,xn} denotes the environment derived from E by "undoing" the bindings of the xi. The union of two environments E and E' with disjoint domains is denoted E + E'.

When we write f(x1,...,xn) = s, we really mean that that strategy definition is in the Stratego program currently under consideration. So evaluation is always *relative to a given Stratego program.*

Two features of Stratego's operational semantics are worth calling out for special attention. First, the relation => is deterministic, i.e., for any strategy s

$$\texttt{id @ p => p} \qquad\qquad \texttt{fail @ p => F}$$

$$\frac{\texttt{s1 @ p => p'' s2 @ p'' => p'}}{\texttt{(s1; s2) @ p => p'}} \qquad \frac{\texttt{s @ p => F}}{\texttt{not(s) @ p => p}}$$

$$\frac{\texttt{s1 @ p => p'}}{\texttt{(s1 <+ s2) @ p => p'}} \qquad \frac{\texttt{s1 @ p => F s2 @ p => p'}}{\texttt{(s1 <+ s2) @ p => p'}}$$

$$\frac{\texttt{f(x1,...,xn) = s s[x1:=s1,...,xn:=sn] @ p => p'}}{\texttt{f(s1,...,sn) @ p => p'}}$$

$$\frac{\texttt{s[x:=rec x(s)] @ p => p'}}{\texttt{rec x(s) @ p => p'}}$$

$$\frac{\texttt{E' > E E'(pat) = t dom(E') = dom(E) + (Var(pat)-dom(E))}}{\texttt{?pat @ (t, E) => (t, E')}}$$

$$\frac{\texttt{E(x) = t'}}{\texttt{!x @ (t, E) => (t', E)}} \qquad \frac{\texttt{(!pat; s) @ p => p'}}{\texttt{!<s>pat @ p => p'}}$$

$$\frac{\texttt{!pat1 @ (t, E0) => (t1,E1) ... !patn @ (t, En-1) => (tn,En)}}{\texttt{!C(pat1,...,patn) @ (t, E0) => (C(t1,...,tn),En)}}$$

$$\frac{\texttt{s @ (t, E-}\{\texttt{x1,...,xn}\}\texttt{) => (t', E')}}{\{\texttt{x1,...,xn : s}\}\texttt{ @ (t,E) => (t', E'-}\{\texttt{x1,...,xn}\}\texttt{ + }\{\texttt{...xi:=E(xi)...}\}\texttt{)}}$$

$$\frac{\texttt{s @ (t, E) => (t', E')}}{\texttt{where(s) @ (t, E) => (t, E')}}$$

$$\frac{\texttt{s @ (t1, E0) => (t1', E1) .... s @ (tn, En-1) => (tn', En)}}{\texttt{all(s) @ (C(t1,...,tn), E0) => (C(t1',...,tn'), En)}}$$

$$\frac{\texttt{s1 @ (t1, E0) => (t1', E1) .... sn @ (tn, En-1) => (tn', En)}}{\texttt{C(s1,...,sn) @ (C(t1,...,tn), E0) => (C(t1',...,tn'), En)}}$$

Figure 7: Operational semantics of Stratego (subset)

and any term-environment pair `p`, there is exactly one possible pair which can result from application of `s` to `p`. As a consequence, `=>` is necessarily confluent.

Second, Stratego's operational semantics is structural, i.e., is compositional. As a result, standard, syntax-directed induction techniques can be used to prove properties of the relation `=>`. We will use such techniques in Sections 7.4 and 8 to prove that our transformations for optimizing applications of `innermost` are correct with respect to the operational semantics of Stratego.

7. Stratego Transformations in Stratego

In the next two sections we discuss and implement a transformation on Stratego programs which fuses applications of `innermost` with the rules to which it is applied. In this section we introduce the basic transformation rules that are used in the transformation and prove their correctness. In the next section we present the strategy that is used to apply these rules and prove it correct. The strategy transformations used to optimize applications of `innermost` will themselves be implemented as Stratego strategies.

7.1. Concrete Syntax

As discussed in Section 3, we use the concrete syntax of Stratego defined in Figure 6 in specifying the program transformations. In order to distinguish the syntax of *object-expressions* that are the subjects of the transformations from the *meta-expressions* that implement transformations we use quotations. That is, `Term|[ pat ]|` indicates a term pattern being transformed, and `Strategy|[ s ]|` represents a strategy expression. Where no ambiguities arise we leave out the `Term` and `Strategy` prefixes.

7.2. Correctness of Transformations

The correctness of each optimizing transformation will be proved when the transformation is introduced. Informally, a strategy transformation is said to be correct if the meaning of the input strategy to the transformation is the same as the meaning of the transformed strategy. Depending on the application, meaning can be assigned to strategies in many different ways. In the study of strategy transformations, it is the operational behavior of strategies that is of interest.

We can formalize the notion of correctness as follows. Let `s1` and `s2` be strategy expressions. The transformation of `s1` to `s2` is *correct* if, for all pairs `p` and `p'`, `s1 @ p => p'` iff `s2 @ p => p'`.

Since Stratego's semantics is compositional, applying a meaning-preserving transformation to a strategy "in context" — i.e., other than at the root of the object language term which represents it — also preserves the meaning of that strategy. Any sequence of meaning-preserving strategy transformations out of any strategy term is therefore meaning-preserving as well. Moreover, since strategies are just ways to apply transformation rules to parts of terms in context, once we establish the correctness of the individual rules that are applied to optimize innermost strategies, we are assured that any strategy used to apply them to various parts of terms is also correct.

7.3. Preprocessing and Desugaring

Some preprocessing and desugaring takes place before transformation begins. The preprocessing consists of renaming various identifiers, and the desugaring expands rules according to their definitions as scoped match-build strategies.

```
module fusion-rules
imports stratego
rules

  AssociateR :
    |[ (s1; s2); s3 ]| -> |[ s1; (s2; s3) ]|

  IntroduceApp :
    |[ !pat; s ]| -> |[ !<s> pat ]|

  AppToSeq :
    Term|[ <s1> (<s2> pat) ]| -> Term|[ <s2; s1> pat ]|

  SeqOverLChoiceL :
    |[ s1; (s2 <+ s3) ]| -> |[ (s1; s2) <+ (s1; s3) ]|

  seq-over-choice =
    ?|[ s3 ]|;
    rules(
      SeqOverLChoiceR :
        |[ (s1 <+ s2); s3 ]| -> |[ (s1; s3) <+ (s2; s3) ]|
    )

  SeqOverScopeR :
    |[ {xs : s1}; s2 ]| -> |[ {xs : s1; s2} ]|

  SeqOverScopeL :
    |[ s1; {xs : s2} ]| -> |[ {xs : s1; s2} ]|

  BottomupOverConstructor :
    Term|[ <bottomup(s)> c(ts1) ]| -> Term|[ <s> c(ts2) ]|
    where <map(\ t -> Term|[ <bottomup(s)> t ]| \ )> ts1 => ts2
```

Figure 8: Distribution and association rules.

Preprocessing and desugaring clearly preserve the meanings of strategies. For example, the correctness of strategy inlining follows from the semantics of strategy definitions given in Figure 7.

7.4. The Transformation Rules

The rules used in the derivation in Section 5 are formalized as Stratego rules in Figure 8. Here we prove the correctness of each rule. In the next section we combine these rules into a strategy that optimizes occurrences of the `innermost` strategy in Stratego specifications.

Associate Composition Right

The rule `AssociateR` associates sequential composition of strategies to the right.

PROPOSITION 7.1: *For all pairs* `p` *and* `p'`,

`((s1; s2); s3) @ p => p'` *iff* `(s1; (s2; s3)) @ p => p'`.

Proof: If `((s1; s2); s3) @ p => p'`, then there must exist a pair `p1` such that `(s1; s2) @ p => p1` and `s3 @ p1 => p'`. The former in turn implies the existence of a pair `p2` such that `s1 @ p => p2` and `s2 @ p2 => p1`. But then `s2 @ p2 => p1` and `s3 @ p1 => p'` imply `(s2; s3) @ p2 => p'`. Together with the fact that `s1 @ p => p2`, this gives `(s1; (s2; s3)) @ p => p'` as desired. The converse is similar. □

Introduce Application

The `IntroduceApp` rule formalizes the assertion in Section 4.1 that `<s> pat` is shorthand for `!pat; s`.

PROPOSITION 7.2: *For all pairs* `p` *and* `p'`,

`(!pat; s) @ p => p'` *iff* `(!<s> pat) @ p => p'`.

Proof: This follows immediately from the semantics of `<s> pat` in Figure 7. □

Application to Sequence

The rule `AppToSeq` replaces an application of a strategy to an application of another strategy with the sequential composition of the two strategies. According to Figure 6, the only place a pattern `pat` can appear in a strategy is in a build pattern. But then from the syntax of build patterns we see that `pat` must appear in some term pattern context. To prove correctness of `AppToSeq`, we must therefore show that the effects of its left- and right-hand sides in context are the same.

PROPOSITION 7.3: *For all pairs* `p` *and* `p'` *and any term pattern context* `Con[.]`

`!Con[<s1> (<s2> pat)] @ p => p'` *iff* `!Con[<s2; s1> pat] @ p => p'`.

Proof: By induction over the context `Con[.]`. First assume the context is empty. We have `(!<s1> (<s2> pat)) @ p => p'` iff `(!<s2> pat; s1) @ p => p'`. But the latter holds iff there exists a `p1` such that `(!<s2> pat) @ p => p1` and `s1 @ p1 => p'`. By Proposition 7.2, `(!<s2> pat) @ p => p1` holds iff `(!pat; s2) @ p => p1`, i.e., iff there exists a pair `p2` such that `!pat @ p => p2` and `s2 @ p2 => p1`. But the latter holds iff `s2 @ p2 => p1` and `s1 @ p1 => p'`, which in turn hold iff `(s2; s1) @ p2 => p'`. Thus `(!<s1> (<s2> pat)) @ p => p'` holds iff `!pat @ p => p2` and `(!pat; (s2; s1)) @ p => p'`, i.e., iff `(!<s2; s1> pat) @ p => p'`.

Now assume that the proposition has been proven for contexts of depth n. If `Con[.]` is a context of depth $n+1$, then it must be of the form `C(..., Con'[.], ...)` with `Con'[.]` a context of depth n. Hence, `!Con[<s1> (<s2> pat)] @ p => p'` iff `!C(pat1,..,Con'[<s1> (<s2> pat)],...,patn) @ p => p'`. But this is the case iff `p` is of the form `(C(t1,...,tn),E0)`, `p'` is of the form `(C(t1',...,tn'),En)`, and, for each argument term `ti`, `!pati @ (ti, Ei-1) => (ti', Ei)`. In particular, this happens iff `!Con[<s1> (<s2> pat)] @ (ti, Ei-1) => (ti', Ei)`. By induction, this is the case iff `!Con[<s2; s1> pat] @ (ti, Ei-1) => (ti', Ei)`, which in turn holds iff `!C(pat1,..,Con'[<s2; s1> pat],...,patn) @ p => p'`, i.e., iff `!Con[<s2; s1> pat] @ p => p'`. □

Sequence over Choice (Left)

The rule `SeqOverLChoiceL` distributes sequential composition on the left over the deterministic choice of two strategies.

Proposition 7.4: *For all pairs* `p` *and* `p'`,

`(s1; (s2 <+ s3)) @ p => p'` *iff* `((s1; s2) <+ (s1; s3)) @ p => p'`.

Proof: If `(s1; (s2 <+ s3)) @ p => p'`, then there exists some `p''` such that `s1 @ p => p''` and `(s2 <+ s3) @ p'' => p'`. Since `(s2 <+ s3) @ p'' => p'`, either `s2 @ p'' => p'` or else `s2 @ p'' => F` and `s3 @ p'' => p'`. In the first case, `s1 @ p => p''` and `s2 @ p'' => p'`, so that `(s1; s2) @ p => p'`. In the second, `s1 @ p => p''` and `s2 @ p'' => F` and `s3 @ p'' => p'`, so that `(s1; s2) @ p => F` and `(s1; s3) @ p => p'`. In either case we have `((s1; s2) <+ (s1; s3)) @ p => p'`.

If `((s1; s2) <+ (s1; s3)) @ p => p'`, then either `(s1; s2) @ p => p'`, or else `(s1; s2) @ p => F` and `(s1; s3) @ p => p'`. In the first case, there must exist a pair `p''` such that `s1 @ p => p''` and `s2 @ p'' => p'`. But the latter implies that `s1 @ p => p''` and `(s2 <+ s3) @ p'' => p'`, so that we have `(s1; (s2 <+ s3)) @ p => p'` as desired.

In the second case, `(s1; s2) @ p => F` and there must exist a `p''` such that `s1 @ p => p''` and `s3 @ p'' => p'`. Since `(s1; s2) @ p => F`, it must be the case that `s2 @ p'' => F`. Thus, `(s2 <+ s3) @ p'' => p'`. Since `s1 @ p => p''` and `(s2 <= s3) @ p'' => p'`, we may conclude that `(s1; (s2 <+ s3)) @ p => p'` in this case as well. □

Sequence over Choice (Right)

The rule `SeqOverLChoiceR` distributes sequential composition on the right over the deterministic choice of two strategies. As discussed in Section 5.3, it is valid only when `s3` is guaranteed to succeed. This is reflected in the implementation in Figure 8, by means of the dynamic rules mechanism. The `seq-over-choice` strategy should be applied for a strategy expression which is known to always succeed. It matches the strategy expression against `s3`, and then the *dynamic rule* `SeqOverLChoiceR` is generated for this specific strategy. Here we prove the correctness of the generated rule, assuming that `s3` is guaranteed to succeed.

Proposition 7.5: *For all pairs* `p` *and* `p'`*, if* `s3` *always succeeds, then*

`((s1 <+ s2); s3) @ p => p'` *iff* `((s1; s3) <+ (s2; s3)) @ p => p'`.

Proof: If `((s1 <+ s2); s3) @ p => p'`, then there must exist a pair `p''` such that `(s1 <+ s2) @ p => p''` and `s3 @ p'' => p'`. The former entails that either `s1 @ p => p''`, or else `s1 @ p => F` and `s2 @ p => p''`. In the first case we have `(s1; s3) @ p => p'`; in the second, `(s1; s3) @ p => F` and `(s2; s3) @ p => p'`. In either case, `((s1; s3) <+ (s2; s3)) @ p => p'`.

If `((s1; s3) <+ (s2; s3)) @ p => p'`, then either `(s1; s3) @ p => p'`, or else `(s1; s3) @ p => F` and `(s2; s3) @ p => p'`. In the first case, there must exist a `p''` such that `s1 @ p => p''` and `s3 @ p'' => p'`. Thus, we have `(s1 <+ s2) @ p => p''`, and so `((s1 <+ s2); s3) @ p => p'`.

In the second case, either `s1 @ p => F`, or else `s1 @ p => p'''` holds and `s3 @ p''' => F`. But `s3` always succeeds by hypothesis, so we must have that `s1 @ p => F`. Furthermore, `(s2; s3) @ p => p'` implies that there exists a pair `p''` such that `s2 @ p => p''` and `s3 @ p'' => p'`. We therefore have `(s1 <+ s2) @ p => p''`, and therefore `((s1 <+ s2); s3) @ p => p'` as desired. □

Sequence over Scope (Right)

The rule `SeqOverScopeR` extends a scope to include a strategy which is composed with it on the right.

Proposition 7.6: *For all pairs* `p` *and* `p'`*,*

`({xs : s1}; s2) @ p => p'` *iff* `{xs : s1; s2} @ p => p'`.

Proof: By renaming, we can assume that the variables `xs` occur only in `s1`. Thus `{xs : s1}; s2` and `{xs : s1; s2}` both have the effect of delimiting the scope of `xs` to `s1` in the composition `s1; s2`, and so both behave as the strategy `s1; s2` on any input pair. □

Sequence over Scope (Left)

The rule `SeqOverScopeL` extends a scope to include a strategy which is composed with it on the left.

Proposition 7.7: *For all pairs* `p` *and* `p'`*,* `(s1; {xs : s2}) @ p => p'` *iff* `{xs : s1; s2} @ p => p'`.

Proof: By renaming, we can assume that the variables `xs` occur only in `s2`. Thus `s1; {xs : s2}` and `{xs : s1; s2}` both have the effect of delimiting the scope of `xs` to `s2` in the composition `s1; s2`, and so both behave as the strategy `s1; s2` on any input pair. □

Bottomup over Constructor

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. Thus, the rule transforms a build pattern of the form `<bottomup(s)> c(pat1,...,patn)` to the build pattern `<s> c(<bottomup(s)> pat1,...,<bottomup(s)> patn)`. In this transformation we assume that there are no pattern matches in the constructor pattern, nor in the strategy `s`.

PROPOSITION 7.8: *Suppose no pattern matches over free variables are done by* `s` *or any* `pati`*. Then for all pairs* `p` *and* `p'`*, and any term pattern context* `Con[.]`*,*

`!Con[<bottomup(s)> c(pat1,...,patn)] @ p => p'` *iff*
`!Con[<s> c(<bottomup(s)> pat1,...,<bottomup(s)> patn)] @ p => p'`.

Proof: By induction on the depth of the context. We give the proof only for the base case — when `Con` is the empty context — since the inductive case is similar to that for `AppToSeq`.

By the definition of `bottomup`, we have `!<bottomup(s)> c(pat1,...,patn) @ p => p'` iff `!<all(bottomup(s)); s> c(pat1,...,patn) @ p => p'`. But this holds iff `(!c(pat1,...,patn); all(bottomup(s)); s) @ p => p'`, i.e., iff there exist pairs `p1` and `p2` such that `!c(pat1,...,patn) @ p => p1`, and `all(bottomup(s)) @ p1 => p2`, and `s @ p2 => p'`. Now, let `p` be `(t,E)`. Then `!c(pat1,...,patn) @ (t,E) => (c(t1,...,tn),E)` iff for each sub-pattern `pi`, we have `!pati @ (t,E) => (ti,E)`. (Here, `E` is constant throughout by the assumption that `pati` does not bind any variables.) But this holds iff `p1` is precisely `(c(t1,...,tn),E)`. By the semantics of the operator `all`, this happens iff `all(bottomup(s)) @ (c(t1,...,tn),E) => (c(t1',...,tn'),E)`, where `bottomup(s) @ (ti,E) => (ti',E)` for each `i`, and therefore `p2` is precisely `(c(t1',...,tn'),E)`. But this is the case iff `(!pati; bottomup(s)) @ (t, E) => (ti',E)`, i.e., iff `(!<bottomup(s)> pati) @ (t, E) => (ti',E)`, which is the case iff `!c(<bottomup(s)> pat1,...,<bottomup(s)> patn) @ p => p1`. Since this holds iff `!<s> c(<bottomup(s)> pat1,...,<bottomup(s)> patn) @ p => p'`, the correctness of `BottomUpOverConstructor` is proved in the base case. □

8. The Transformation Strategy

In the derivation in Section 5 we implicitly used a particular strategy to apply the rewrite rules which optimize applications of `innermost`. More specifically, we applied the rules to certain subterms in a certain order. In this section we make explicit the strategy we used, show how it can be coded in Stratego, and argue that the optimized strategies resulting from its application have the same observable behavior as the ones from which they are derived.

8.1. Fusion Strategy

The overall strategy employed is the strategy `fusion` in Figure 9. The strategy transforms a complete Stratego specification, and is composed of three main substrategies:

1. `declare-inline-rules` which generates, for each strategy operator definition, a rewrite rule that inlines calls to that strategy operator.
2. `check-library-definitions`, which verifies that the library definitions of certain strategy operators conform to their expected semantics.
3. `innermost-fusion`, which uses a top-down traversal to fuse all calls to `innermost`.

We consider these substrategies in detail in the next three subsections. Note that the `iowrap(s)` combinator turns a transformation `s` into a program that can deal with command-line options and input/output of terms.

Our goal in this section is to argue that `fusion` preserves meanings of specifications:

PROPOSITION 8.1: *For any strategy definition* `f(as) = s` *in a specification, if* `s` *is transformed to* `s'` *by* `fusion`*, then for all pairs* `p` *and* `p'`*,* `s @ p => p'` *iff* `s' @ p => p'`*.*

We do this by arguing that each of its constituent strategies `declare-inline-rules`, `check-library-definitions`, and `alltd(innermost-fusion)` preserves the meanings of strategies, i.e., is correct.

```
fusion =
  iowrap(
    declare-inline-rules
    ; check-library-definitions
    ; alltd(innermost-fusion)
  )
```

Figure 9: Fusion strategy.

```
declare-inline-rules =
  Specification([Signature(id),
                 Strategies(map(declare-inline-rule))])

declare-inline-rule =
  ?|[ f(as) = s ]|;
  rules(
    InlineStrat :
      Strategy|[ f(ss) ]| -> <ssubs> (as', ss, s')
      where <strename> |[ f(as) = s ]| => |[ f(as') = s' ]|
  )

inline-rules =
  rec x(try(
       LChoice(x, x)
       <+ Scope(id, Seq(Match(id),Build(id)))
       <+ Scope(id, Seq(Match(id),Seq(id,Build(id))))
       <+ InlineStrat; x
  ))
```

Figure 10: Inline strategy definitions

8.2. Inlining

Initially, the argument to `innermost` is a deterministic choice of rules. In order to specialize `innermost` to a choice of rules, each rule's definition must first be desugared into its scoped match-build representation. This ensures that the innermost strategy to be optimized is in the right form for processing by `inline-rules`.

8.2.1. Generating Inlining Rules

Figure 10 defines the strategy `declare-inline-rules` which generates, for each strategy definition in a specification, an inlining rule `InlineStrat`. This is done by mapping `declare-inline-rule` over the list of strategy definitions in a specification. The strategy `declare-inline-rule` matches a strategy definition and then generates an `InlineStrat` rule specific for that definition. Since each such rule is generated dynamically, it inherits the bindings of the variables `f`, `as`, and `s` from the context of its corresponding strategy definition.

Each generated inlining rule matches a strategy application `f(s1,...,sn)` and replaces it with the body of the strategy definition of `f` in which the formal parameters `a1,...,an` are replaced with the actual parameters `s1,...,sn`. In order to prevent name capture, the original strategy definition is renamed using `strename`, which renames bound strategy and term pattern variables. Substitution of actuals for formals is then achieved using the substitution strategy `ssubs`.

Proposition 8.2: *For each* `InlineStrat` *rule generated by an application of* `declare-inline-rule` *to a strategy definition* `f(a1,...,an) = s`*, if* `s` *is the result of applying* `InlineStrat` *to the strategy expression* `f(s1,...,sn)`*, then*

`f(s1,...,sn) @ p => p'` *iff* `s @ p => p'`

Proof: Since `s` corresponds to the body of `f` in which formal parameters have been replaced by actual parameters, this follows immediately from the semantics of strategy calls. □

Correctness of `declare-inline-rules` follows immediately from the following proposition.

Proposition 8.3: *The strategy* `declare-inline-rules` *(1) is the identity transformation on specifications and (2) generates for each strategy definition a meaning preserving inlining rule* `InlineStrat`*.*

Proof: (1) Neither the congruence strategy in `declare-inline-rules`, the match against the strategy definition in `declare-inline-rule`, nor the `rules` constructed in `declare-inline-rule` changes the term to which it is applied. (2) The `declare-inline-rule` is applied to each strategy definition in turn by `map` and, according to the previous proposition, generates a correct inlining rule. □

8.2.2. Using Inlining Rules

After inlining rules have been generated, they can be applied anywhere without changing the meanings of programs. However, replacing every call everywhere is not a useful strategy from the point of view of code size, and there is also the danger of non-termination of the inliner when expanding recursive calls. Inlining is, therefore, best applied selectively. In the case of the current transformation, only the rules passed as arguments to the `innermost` strategy are inlined.

The `inline-rules` strategy inlines calls in a strategy expression, but only top-level calls and calls in arguments to the deterministic choice operator are inlined. In particular, inlining stops when a (desugared) rule is encountered. Moreover, calls which are embedded in `where` clauses in rules are not inlined. However, whenever a strategy call is inlined, `inline-rules` is applied recusively to the instantiated body of that strategy. This properly handles definitions of the form

```
simplify = innermost(rule-set1 <+ rule-set2)
```

where

```
rule-set1 = A1 <+ ... <+ An
rule-set2 = B1 <+ ... <+ Bm
```

in which abstractions over sets of rules are passed as arguments to `innermost`.

```
check-library-definitions =
  check-that-try-is-try
  ; check-that-innermost-is-innermost
  ; check-that-bottomup-is-bottomup

check-that-innermost-is-innermost =
  where(
    new => x
    ; <InlineStrat> Strategy|[ innermost(x()) ]|
    ; ?|[ bottomup(try(x(); innermost(x()))) ]|
  )
```

Figure 11: Check implementations of generic strategies

PROPOSITION 8.4: *The strategy* `inline-rules` *(1) is a meaning preserving transformation on strategy expressions and (2) reduces strategy expressions to the form* `s1 <+ ... <+ sn`*, where none of the* `si` *are strategy calls.*

Proof: (1) As stated by Proposition 8.2, `InlineStrat` is meaning preserving. Since the entire computational effect of `inline-rules` is to apply `InlineStrat` to some subexpressions, it is meaning preserving. (2) The strategy replaces each call residing under a left choice operator by its definition. □

8.3. Verifying Library Definitions

Our innermost fusion strategy is application-specific, i.e., it makes assumptions about the semantics of certain strategy operators in the library. In particular, the specifications of `innermost`, `bottomup`, and `try` are assumed to have (the meaning of) the forms given in Figure 3. We could just make these assumptions and leave it up to the programmer or library writer to make sure this is the case. However, since libraries are subject to change and can be extended, it is safer to build in a check that these assumptions are actually valid.

The strategy `check-library-definitions` verifies that the assumptions are indeed valid using three `check-...` strategies, each of which checks the form of a specific strategy definition. The strategy `check-that-innermost-is-innermost`, for example, first inlines the definition of `innermost` from the library with a newly named strategy as its argument. The strategy resulting from the inlining is then matched against the expected result. The match succeeds iff the library definition of `innermost` has the proper form. The strategies `check-that-try-is--try` and `check-that-bottomup-is-bottomup` perform similar verifications.

Even though it is possible to match against several (semantically equivalent) variants of a strategy definition, this will never account for all possible definitions that implement that particular strategy. Verifying library definitions is thus application-specific. Instead of providing a general program analysis to detect that a definition implements a particular strategy, we use knowledge of the application at hand to enable a transformation.

PROPOSITION 8.5: *The strategy* `check-library-definitions` *(1) is the identity transformation on specifications and (2) when it succeeds in the context of a set of* `InlineStrat` *rules, the definitions for* `try`*,* `innermost`*, and* `bottomup` *have the expected semantics.*

Proof: (1) A `where(s)` strategy does not change the term to which it is applied. (2) The `check-f-is-f` strategies match the definition of `try`, `innermost`, and `bottomup` against their expected definitions, and fail if incompatible definitions are found. □

8.4. Innermost Fusion

The last step of the `fusion` strategy in Figure 9 is the traversal of the specification using a one-pass traversal which tries to apply `innermost-fusion`. The `innermost-fusion` rule defined in Figure 12 consists of three parts. The first part prepares the argument to the call to `innermost` for fusion, the second part performs the fusion of the build with bottomup, and the third part realizes the goal of the transformation, i.e., it prevents the renormalization of variable subterms. It is assumed that desugaring, inlining, and verification have already been performed prior to application of `innermost-fusion`. We therefore assume that we have inlining rules for each strategy definition at our disposal, and that the library functions all have their expected meanings.

8.4.1. Preparing for Fusion

The `innermost-fusion` rule recognizes a call to `innermost` and replaces it with a canonical implementation in which the argument strategy is inlined and fused. The rule first generates a fresh variable using the `new` strategy primitive. This primitive creates a new name that is guaranteed not to occur anywhere in any term being processed by the transformation. Using this fresh variable, the rule replaces `innermost(s1)` by an unfolding of that strategy. The replacement is correct according to the following proposition and the observation that, in `innermost-fusion`, the strategies `s1` and `s2` have the same semantics by Proposition 8.4.

PROPOSITION 8.6: *For any strategy* `s`*, any strategy variable* `x` *not occurring in* `s`*, and any pairs* `p` *and* `p'`*, we have that*

```
                innermost(s) @ p => p' iff
        bottomup(rec x(try(s; bottomup(x)))) @ p => p'
```

Proof: By definition of `innermost` we have that `innermost(s) @ p => p'` iff `bottomup(try(s; innermost(s))) @ p => p'`. By again applying the definition of `innermost` we get

```
innermost-fusion :
  |[ innermost(s1) ]| -> |[ bottomup(rec x(try(s3))) ]|
  where new => x
      ; <seq-over-choice> Strategy|[ bottomup(x) ]|
      ; <bottomup-to-var> Strategy|[ bottomup(x) ]|
      ; <inline-rules> s1 => s2
      ; <fuse-with-bottomup> |[ s2; bottomup(x) ]|
      ; prevent-renormalization => s3
```

Figure 12: The innermost fusion strategy

```
bottomup(try(s; bottomup(try(s; innermost(s))))) @ p => p'
```

The recursive pattern thus emerging can be folded to

```
bottomup(rec x(try(s; bottomup(x)))) @ p => p'
```

Unfolding this strategy leads to the same sequence of transformations as the unfolding of `innermost`. □

Based on knowledge of this local context, the `innermost-fusion` rule then generates dynamic rules. First, the `seq-over-choice` strategy generates an instance of the `SeqOverLChoiceR` rule with `bottomup(x)` as the term to distribute. This is justified according to the following propositions.

PROPOSITION 8.7: *If* `s` *always succeeds, then* `bottomup(s)` *always succeeds, i.e., for any term* `t` *and environment* `E` *there exist a term* `t'` *and environment* `E'` *such that* `bottomup(s) @ (t, E) => (t', E')`.

Proof: By induction on terms. In the base case, `t` must be of the form `C()`. By the semantics of `all`, we have that `all(bottomup(s)) @ (C(),E) => (C(),E)`. By the semantics of sequential composition and the fact that `s` always succeeds, there must exist `t'` and `E'` such that `all(bottomup(s)); s @ (C(),E) => (t',E')`. By the definition of `bottomup`, we have `bottomup(s) @ (C(),E) => (t',E')`.

In the inductive case, `t` must be of the form `C(t1,...,tn)` for terms `t1`, ..., `tn`. By the induction hypothesis, there must exist `t1'` and `E1` such that `bottomup(s) @ (t1, E) => (t1', E1)`, there must exist `t2'` and `E2` such that `bottomup(s) @ (t2, E1) => (t2', E2)`,..., and there must exist `tn'` and `En` such that `bottomup(s) @ (tn, En-1) => (tn', En)`. Thus, by the semantics of `all`, `all(bottomup(s)) @ (C(t1,...,tn), E) => (C(t1',..,tn'), En)`. Now, by the semantics of sequential composition and the fact that `s` always succeeds, there must exist `t'` and `E'` such that `all(bottomup(s)); s @ (C(t1',.., tn'), En) => (t',E')`. Therefore, `bottomup(s) @ (C(t1',..,tn'), E) => (t', E')` by the definition of `bottomup`. □

PROPOSITION 8.8: *In the context of the strategy*

```
fuse-with-bottomup =
  innermost(
    SeqOverLChoiceR
    <+ SeqOverScopeR
    <+ AssociateR
    <+ IntroduceApp
    <+ BottomupOverConstructor
  )
```

Figure 13: Fuse with bottomup

```
bottomup(rec x(try(s; bottomup(x))))
```

the strategy `bottomup(x)` *is guaranteed to succeed.*

Proof: First observe that, for any `s`, `try(s)` is guaranteed to succeed. Indeed, `try(s) = s <+ id`, so that even if the application of `s` fails, `try(s)` still succeeds. In particular, `try(s; bottomup(x))` always succeeds. From this we conclude that `x` always succeeds. Then, by previous proposition, `bottomup(x)` always succeeds. □

Next, the strategy `bottomup-to-var` generates rules that eliminate the application of `bottomup(x)` to variables from the left-hand side of rules. This will be elaborated below.

After these preparatory steps, rule `innermost-fusion` first inlines rules in the argument `s1` of `innermost`. This results in a strategy `s2` which is equivalent to `s1` by Proposition 8.4. Subsequently these inlined rules are fused with `bottomup(x)`, and finally, the applications of `bottomup(x)` to variables from the left-hand sides of rules are eliminated by `prevent-renormalization`.

8.4.2. Performing Fusion

The fusion of the right-hand sides of the rules with the `bottomup(x)` strategy is implemented by the `fuse-with-bottomup` strategy in Figure 13. Together, the rules `SeqOverScopeR`, `AssociateR`, `IntroduceApp`, `BottomupOverConstructor`, and `SeqOverLChoiceR` have the effect of applying the inner occurrence of `bottomup` to each variable occurrence in the right-hand side of each normalizing rule. This transformation is correct by the correctness of the individual transformation rules; Proposition 8.8 ensures the correctness of `SeqOverLChoiceR` in this instance.

8.4.3. Preventing Renormalization

The final step of the `fusion` transformation is the transformation of applications of `bottomup` of the form `<bottomup(x)>y` to just `y` whenever `y` is a variable occurring in the left-hand side of one of the normalization rules, and the application

```
prevent-renormalization =
  apply-to-rules(BottomupToVarIsId-UnCond
                <+ BottomupToVarIsId-Cond)

apply-to-rules(s) =
  rec x(try(LChoice(x, x) <+ Scope(id, x) <+ s))

bottomup-to-var = ?bu;
  rules(
    BottomupToVarIsId-UnCond :
      |[ ?t1; !t2 ]| -> |[ ?t1; !t3 ]|
      where <replace-application> (bu, t1, t2) => t3

    BottomupToVarIsId-Cond :
      |[ ?t1; where(s); !t2 ]| -> |[ ?t1; where(s'); !t3 ]|
      where <replace-application> (bu, t1, (s, t2)) => (s', t3)
  )

 replace-application :
   (s, t1, t2) -> t3
   where {| Replace :
     <tvars> t1
     ; map({?x; rules(Replace : Term|[ <s> x ]| -> Term|[ x ]|)})
     ; <alltd(Replace)> t2 => t3
   |}
```

Figure 14: Prevent renormalization

occurs in the right-hand side of the rule. This transformation is implemented by the strategy `prevent-renormalization` in Figure 14. The strategy performs a traversal over the fused argument of `innermost` using the traversal strategy `apply-to-rules(s)`, which applies `s` to the branches of a choice under the scope of a rule. To each rule it applies one of the `BottomupToVarIsId` rules. These rules are generated by `bottomup-to-var` for the specific `bottomup(x)` expression with which the rules were fused.

The actual application replacement is done by the rule `replace-application`. It takes a triple `(s,t1,t2)` of the strategy `s` to be removed, the left-hand side `t1`, and the right-hand side `t2` of the rule. The strategy `tvars` yields the list of variables in a term pattern. Thus, the `map` expression generates for each variable `x` in the left-hand side a `Replace` rule, which replaces an occurrence of `<s> x` with just the variable `x`. These `Replace` rules are then applied to the right-hand side `t2` using the `alltd` traversal yielding the new right-hand side `t3`.

8.5. Correctness

So far we have shown all steps correct except the last `prevent-renormalization` one. What remains to be proven is that is valid to replace `<bottomup(x)> y` by `y`, if `y` is a variable from the left-hand side of one of the argument rules of `innermost`. What we need to prove is that `bottomup(x)` is the identity on terms bound to variables on the left-hand side of a rule, since those terms are already in normal form.

Definition: Given a strategy `s`, we say that `t` is in *s-normal form* if, for all environments `E`, and each subterm `t'` of `t`, we have `s @ (t', E) => F`, i.e., `s` does not apply to any subterm.

PROPOSITION 8.9: *If a term* `t` *is in* `s`*-normal form, then*

```
innermost(s) @ (t,E) => (t,E)
```

that is, `innermost(s)` *is the identity transformation on terms in* `s`*-normal form.*

Proof: In the base case, when `t` is a nullary constructor application `c`, we reason as follows:

```
                 innermost(s) @ (c,E) => (c,E)
    iff all(innermost(s)); try(s; innermost(s)) @ (c,E) => (c,E)
              iff try(s; innermost(s)) @ (c,E) => (c,E)
```

Since `c` is in `s`-normal form, we have that `s @ (c,E) => F`. Therefore, we have that `try(s; innermost(s))` succeeds by `id`, i.e., `id @ (c,E) => (c,E)` as desired.

In the inductive case, `t` is `c(t1,...,tn)` and we assume that, for all smaller terms than `t`, the proposition holds. Then

```
                 innermost(s) @ (t,E) => (t,E)
    iff all(innermost(s)); try(s; innermost(s)) @ (t,E) => (t,E)
                 iff there exist t' and E' such that
               all(innermost(s)) @ (t,E) => (t',E')
            and try(s; innermost(s)) @ (t',E') => (t,E)
```

But `all(innermost(s)) @ (t,E) => (t',E')` holds iff

```
            innermost(s) @ (t1,E) => (t1',E1),
           innermost(s) @ (t2,E1) => (t2',E2),
                            ...
          innermost(s) @ (tn,En-1) => (tn',E')
```

By the induction hypothesis,

innermost(s) @ (t1,E) => (t1,E),
innermost(s) @ (t2,E1) => (t2,E1),
...
innermost(s) @ (tn,En-1) => (tn,En-1)

so that `ti'` is precisely `ti` for each `i`, and `E = E1 = E2 = .... = En-1 = E'`. Thus `t` and `t'` must be identical, so that

`try(s; innermost(s)) @ (t',E') => (t, E)`
iff `try(s; innermost(s)) @ (t,E) => (t, E)`
iff `id @ (t,E) => (t, E)` since `t` is in `s`-normal form

From this we conclude that `innermost(s) @ (t,E) => (t,E)` holds if and only if `all(innermost(s)) @ (t,E) => (t,E)`. Since we saw above that the latter holds, the theorem is proved. □

COROLLARY 8.1: *If* `t` *is in* `s`*-normal form then*

`bottomup(innermost(s)) @ (t,E) => (t,E),`

that is `bottomup(innermost(s))` *is the identity transformation on terms in* `s`*-normal form.*

Proof: Since `bottomup(innermost(s))` just applies `innermost(s)` to subterms of its subject term, Proposition 8.9 ensures that it is the identity on terms in `s`-normal form. □

PROPOSITION 8.10: *If* `innermost(s) @ (t,E) => (t', E')` *then* `t'` *is in* `s`*-normal form, that is,* `innermost(s)` *reduces terms to* `s`*-normal form.*

Proof: By induction on the depth of the derivation which witnesses the fact that `innermost(s) @ (t, E) => (t', E')`. In the base case we must consider the shortest such derivation. This occurs when `t` is a nullary constructor `c`, and `s @ (c,E) => F`. Then `all(innermost(s)) @ (c,E) => (c,E)` by the semantics of `all`, and `try(s; innermost(s)) @ (c,E) => (c,E)`. (This is because `try(s; innermost(s)) @ (c,E)` must be `id @ (c,E)` in this case.) By the semantics of sequential composition we have `innermost(s) @ (c,E) => (c, E)`.

For the inductive case, assume that `innermost(s) @ (t,E) => (t', E')` holds, and that the proposition has been proven for all derivations shorter than that of `innermost(s) @ (t,E) => (t', E')`. Then we reason as follows:

`innermost(s) @ (t,E) => (t', E')`
iff `all(innermost(s)); try(s; innermost(s)) @ (t,E) => (t', E')`
iff there exist `t''` and `E''` such that
`all(innermost(s)) @ (t,E) => (t'', E'')`
and `try(s; innermost(s)) @ (t'',E'') => (t', E')`

Let `t` be `c(t1,...,tn)`. Then by semantics of `all`, there must be terms `t1'`, ..., `tn'` and environments `E1`, ..., `En` such that

```
innermost(s) @ (t1,E) => (t1',E1)
...
innermost(s) @ (tn,En-1) => (tn',En)
```

By the induction hypothesis we have that the terms `t1'`, ..., `tn'` are in `s`-normal form. Hence, `t''` is `c(t1',...,tn')` and `E''` is `En`.

We now distinguish two cases: We either have `s @ (t'', E'') => F` or else `s @ (t'', E'') => (t''',E''')` for some `t'''` and `E'''`. In the first case, `try(s; innermost(s)) @ (t'',E'') => (t'',E'')`, so that `t''` is `t'` and `E''` is `E'`. But then `t' = c(t1',...,tn')` with each `ti'` in `s`-normal form, `s @ (t',E') => F`. Since each proper subterm of `t'` is in `s`-normal form, and since `s` does not apply to `t'`, we must have that `t'` is itself in `s`-normal form.

In the second case, we have that `innermost(s) @ (t''',E''') => (t',E')`. By the induction hypothesis, `t'` must be in `s`-normal form. □

COROLLARY 8.2: *If*

```
all(innermost(s)) @ (c(t1,...,tn),E0) => (c(t1',...,tn'), En)
```

then all `ti'` *are in* `s`*-normal form.*

Proof: Immediately by the semantics of `all` and previous proposition. □

PROPOSITION 8.11: *If, in an application of the form* `innermost(s) @ (t,E)`*, the strategy* `s` *is applied to the root of a term* `t'`*, then all proper subterms of* `t'` *are in* `s`*-normal form.*

Proof: If, in an application of the form `innermost(s) @ (t,E)`, `s` is applied to the root of a term `t'`, then there must exist `t''`, `E''`, `t3`, `E3`, `E'`, `c`, and `t1'`, ..., `tn'` such that `t'` is precisely `c(t1',...,tn')`, and `all(innermost(s)) @ (t'',E'') => (t',E')` and `try(s; innermost(s)) @ (t',E') => (t3,E3)`. By Corollary 8.2, each `ti'` is in `s`-normal form, i.e., each proper subterm of `t'` is in `s`-normal form. □

COROLLARY 8.3: *In an application of the form* `innermost(s) @ (t,E)`*, and if* `s` *is a choice of the form* `s1 <+ ... <+ sn` *and* `si` *is a rule of the form* `{xs: ?t1; !t2}`*, then the terms bound to variables in* `?t1` *are in* `s`*-normal form, and the recursive application of* `innermost(s)` *to these terms in the right-hand side* `!t2` *is the identity transformation.*

Hence, the replacement of the application `<bottomup(x)> y` by `y` for variables `y` that are bound in the left-hand side of a rule is valid.

Corollary 8.4: *Assuming that* `InlineStrat` *rules are available for all definitions in the Stratego program at hand and assuming that library definitions for* `try`*,* `bottomup`*, and* `innermost` *have the expected semantics, then the transformation rule* `innermost-fusion` *transforms an application of* `innermost` *into an equivalent strategy expression.*

Corollary 8.5: *The* `fusion` *transformation is a meaning preserving transformation on Stratego specifications.*

9. Conclusion and Discussion

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

Our approach makes use of a number of interesting transformation techniques and idioms. We avoid the manipulation of large and unwieldy abstract syntax trees by specifying the transformation rules on which our optimization is based in terms of the concrete syntax of the object language. In addition, our optimizing strategy relies heavily on Stratego's ability to generate rewrite rules at run-time. Since these dynamic rules can access information available from their generation contexts, they allow the specification of transformations that would not be possible to express with standard, context free rewrite rules alone.

Since our transformation strategies are based on rewriting, they necessarily perform their work using cascading transformations, i.e., by applying in combination a number of small, independent transformations to achieve a large-scale effect. But to limit the availability of certain such transformations to specific parts of computations, we also use the idiom of staged transformation to break our overall transformation into several phases.

Like staged computations, local transformations also allow us to limit the applicability of transformation rules, albeit in a different way. A local transformation uses special knowledge about its subject program at the point of application to perform computations that would not be beneficial if applied throughout the entire program. Unlike staged transformations, local transformations are often available throughout a computation, even though they are applicable to a subject term only when certain criteria are met.

Application-specific transformations take the use of special knowledge one step further, allowing certain information about a specific library or an entire specification to be used in transformation. We use application-specific transformations in our optimization of innermost normalization to ensure that library functions have appropriate forms and semantics.

Stratego supports all six of these transformation idioms. The optimization strategy presented in this paper, which relies on them, is included as an optimization phase in the Stratego compiler (version 0.9).

The specialization of the `innermost` strategy with the rules it is applied to, decreases the complexity from $O(n^2)$ to $O(n)$ in the number of rewrite steps, thus reaching up to 2 million rewrites per second for the example rewrite system. It should be noted that for rules with a more complex left-hand side the cost per rewrite step is higher. However, the efficiency improvement with respect to the unoptimized case is dramatic.

9.1. Previous Work

This paper is an elaboration of an earlier paper [11] on the implementation of innermost fusion. Since that earlier paper appeared in print, Stratego has been extended with dynamic rules and conrete syntax, both of which are used extensively in the specification presented here. Use of these extensions has greatly improved and clarified the specification.

Strategies have also been used to optimize programs which are not themselves defined in terms of strategies. In [10], for example, they are used to eliminate intermediate data structures from functional programs. In [27], 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.

9.2. Related Work

9.2.1. Staged, Cascading Transformation

Small local transformations have been dubbed “humble transformations” in [20]. Such transformations are used extensively in optimizing compilers based on the compilation-by-transformation idiom [13, 14, 1, 19]. They are also used to some degree in most compilers, although not necessarily recognizable as rewrite rules in the implementation. The advantage of the programmable strategies approach is that transformation steps can indeed be formulated as separate rewrite rules, which can be combined into optimization phases by means of strategies.

9.2.2. Traversal Optimization

The optimization of `innermost` presented in this paper was inspired by more general work on functional program optimization. In [9], 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. 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 and automatable fusion strategy which is provably correct with respect to the operational semantics.

The importance of optimizing term traversals in functional transformation systems is discussed in [15]. Term traversals are modelled there by fold functions but, since the fold algebras under consideration are updateable, standard fusion techniques for functional programs [28, 16, 29] 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 [15], 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.

9.2.3. Application-Specific Optimization

While modern programming languages are well suited to *expressing* abstract programs, the compilers for these languages cannot always derive efficient implementations for combinations of abstractions. While generic optimizations can be clever, they are always at a disadvantage with respect to the programmer, who has additional information about the semantics of a program. Therefore, an emerging approach is that of domain-specific or even application-specific optimization, i.e., extending the capabilities of compilers by letting the programmer specify additional information about their programs. Examples are the Broadway Compiler [7], which aims to make software libraries more portable and efficient by allowing them to be automatically customized for different hardware and software environments. User-definable rules in the Glasgow Haskell Compiler GHC [18] allow the programmer to state identities over program expressions that are used by the compiler to simplify programs. In the Simplicissimus project [21] compilers are adapted such that they can make use of semantic information about an ADT at its natural abstraction level. The transformation described in this paper uses the same approach, that is, uses reasoning by the programmer about a specific library definition, i.e., the definition of `innermost` to derive more efficient implementations. While this specification currently is a stage provided by the compiler, in the future we aim to make the Stratego compiler extensible by arbitrary user-definable transformations.

9.3. Future Work

The results of this paper demonstrate that rewriting strategies for optimizing programs can be implemented both effectively and elegantly using local, application-specific transformations. The optimizing strategy presented here fuses innermost normalizations of terms with respect to collections `R1`, ..., `Rn` of standard rewrite rules. Optimizing strategies for applications of bottomup, down-up, and other common term traversals to choices of standard rewrite rules would also be useful.

The rewrite rules with respect to which a term is to be normalized generate side conditions — usually equality of certain subterms modulo normalization — that must be verified in order to assure applicability. In these cases, a version of our innermost fusion strategy which handles applications of `innermost` to choices of such conditional rewrite rules would be useful.

We may also try to extend our fusion technique along both dimensions (traversal and rules) simultaneously. More generally, we may aim to develop a general theory of traversal fusion for Stratego. Additional measurements are needed to evaluate the optimizations achieved by the innermost fusion strategy presented here, and similar benchmarks for any extensions of this strategy will also be in order.

Finally, it would be interesting to know whether mechanical derivation of innermost fusion and its extensions is possible from the definitions of the strategies themselves, as well as to what degree the Stratego compiler itself can be fused.

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

[1] A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] J. M. Boyle. Abstract programming and program transformation—An approach to reusing programs. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, volume 1, pages 361–413. ACM Press, 1989.

[3] J. M. Boyle, T. J. Harmer, and V. L. Winter. The TAMPR program transformation system: Simplifying the development of numerical software. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 353–372. Birkhäuser, 1997.

[4] T. Dinesh, M. Haveraaen, and J. Heering. An algebraic programming style for numerical software and its optimization. *Scientific Programming*, 8(4), 2001.

[5] P. Fradet and D. L. Métayer. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems*, 13(1):21–51, Jan. 1991.

[6] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 223–232. ACM Press, 1993.

[7] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Second Conference on Domain-Specific Languages (DSL'99)*, Austin, Texas, October 1999. USENIX Assocation. `http://www.usenix.org/events/dsl99/guyer.html`.

[8] M. Haveraaen, H. A. Friis, and T. A. Johansen. Formal software engineering for computational modeling. *Nordic Journal of Computing*, 6(3):241–270, 1999.

[9] P. Johann. Short cut fusion: Proved and improved. In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation (SAIG'01)*, volume 2196 of *Lecture Notes in Computer Science*, pages 47–71, Florence, Italy, 2001. Springer-Verlag.

[10] 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.

[11] P. Johann and E. Visser. Fusing logic and control with local transformations: An example optimization. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.

[12] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[13] R. Kelsey and P. Hudak. Realistic compilation by program transformation. In *ACM Conference on Principles of Programming Languages*, pages 281–292, January 1989.

[14] R. A. Kelsey. *Compilation by Program Transformation.* PhD thesis, Yale University, May 1989.

[15] J. Kort, R. Lämmel, and J. Visser. Functional Transformation Systems. In *Proceedings of the 9th International Workshop on Functional and Logic Programming*, Benicassim, Spain, July 2000.

[16] 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.

[17] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[18] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting

as a practical optimisation technique in GHC. In R. Hinze, editor, *2001 Haskell Workshop*, Firenze, Italy, September 2001. ACM SIGPLAN.

[19] S. L. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a techical overview. In *UK Joint Framework for Information Technology (JFIT) Technical Conference*, March 1993.

[20] 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.

[21] S. Schupp, D. Gregor, and D. Musser. User-extensible simplification — type-based optimizer generators. In R. Wilhelm, editor, *Compiler Construction (CC'01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 86–101, Genova, Italy, April 2001. Springer-Verlag.

[22] J. Svenningsson. Shortcut fusion for accumulating parameters and zip-like functions. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pages 124–132. ACM Press, 2002.

[23] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In S. L. Peyton-Jones, editor, *Functional Programming and Computer Architecture (FPCA'95)*, San Diego, California, June 1995.

[24] E. Visser. Scoped dynamic rewrite rules. In M. van den Brand and R. Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, September 2001.

[25] 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)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[26] E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[27] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

[28] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[29] 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.