*Electronic Notes in Theoretical Computer Science 57 (2001)*

*URL:* [*http://www.elsevier.nl/locate/entcs/volume57.htm*](http://www.elsevier.nl/locate/entcs/volume57.html)*l 19 pages*

*Fusing Logic and Control with Local* Transformations: An Example Optimization

*Patricia Johann*

*Department of Mathematics and Computer Science, Dickinson College, Carlisle, PA 17013,* [*http://www.dickinson.edu/*](http://www.dickinson.edu/) *johannp,* [*johannp@dickinson.edu*](mailto:johannp@dickinson.edu)

*Eelco Visser*

*Institute of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands.* [*http://www.cs.uu.nl/*](http://www.cs.uu.nl/) *visser,* [*visser@acm.org*](mailto: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 computa-* *tional overhead it incurs. Fusion techniques can be used to combine the reusability of abstract programs with the eÆciency of specialized programs.*

*In this paper we illustrate some of the ways in which rewriting strategies can be used to separate the de nition of program transformation rules from the strategies under which they are applied. Doing so supports the generic de nition of program transformation components. Fusion techniques for strategies can then be used to specialize such generic components.*

*We show how the generic innermost rewriting strategy can be optimized by fusing it with the rules to which it is applied. Both the optimization and the programs to which the optimization applies are speci ed in the strategy language Stratego. The optimization is based on small transformation rules that are applied locally under the control of strategies, using special knowledge about the contexts in which the rules are applied.*

# *1 Introduction*

*Abstract programming techniques support the generic de nition of algorith-* mic functionality in such a way that di erent con gurations of algorithms can be obtained by plugging together generic components. As a result, these com- ponents can be reused in many instances and in many di erent combinations. The advantages of abstract programming are reduced code size and increased

*c 2001 Published by Elsevier Science B. V. Open access under* [*CC BY-NC-ND license.*](http://creativecommons.org/licenses/by-nc-nd/3.0/)

*reusability of programs. A disadvantage is that the separation of concerns* it supports can introduce considerable computational overhead. In contrast, code for speci c problem instances can e ectively intermingle logic and con- trol to arrive at more eÆcient implementations than are possible generically. The challenge of abstract programming is to maintain a high-level separa- tion of concerns while simultaneously achieving the eÆciency of intermingled programs.

*Fusion techniques mitigate the tension between modularity and eÆciency* by automatically deriving intermingled eÆcient versions of programs from their abstract composite versions. For example, in deforestation of functional programs, intermediate data structures are eliminated by fusing together func- tion compositions [6,11,17]. Fusion also enables transformation from an alge- braic style of programming resembling mathematical speci cation of numeric programs to an updating style in which function arguments are overwritten in order to reuse memory allocated to large matrices [2,4].

*Stratego [15,16] is a domain-speci c language for the speci cation of pro-* gram transformation systems based on the paradigm of rewriting strategies. Stratego separates the speci cation of basic transformation rules from that of the strategies by means of which they are applied. Strategies that control the application of transformation rules can be programmed using a small set of primitive strategy combinators. These combinators support the de nition of very generic patterns of control, allowing strategies and rules to be com- posed as necessary to achieve various program transformations. This abstract programming style leads to concise and reusable speci cations of program transformation systems. However, due to their genericity, some strategies do not have enough information to perform their tasks eÆciently, even though specializations of those strategies could be implemented eÆciently.

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

*In conventional program optimization, transformations are applied through-* out a program. In optimizing imperative programs, complex transformations are applied to entire programs [12]. In the style of compilation by transforma- tion [3] | as applied, for example, in the Glasgow Haskell Compiler [14] | a large number of small, almost trivial program transformations are applied throughout a program to achieve large-scale optimization by accumulating small program changes. The style of optimization that we develop in this pa- per isa combination of these ideas: Combinea number of small transformation steps using strategies that will apply them to speci c parts of a program to achieve the e ects of complex transformations. Because the transformations are local, special knowledge about the subject program at the point of appli-

*cation can be used. This allows the application of rules that would not be* otherwise applicable.

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

# *2 A Generic Speci cation of Innermost Reduction*

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

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

*In Stratego, programs to be transformed are expressed as rst-order terms.* Signatures describe the structure of terms. A term over a signature S is either a nullary constructor C from S or the application C(t1,...,tn) of an n-ary constructor C from S to terms ti over S. For example, Zero, Succ(Zero), and Plus(Succ(Zero),Zero) are terms over the signature in Figure 1.

*2.1 Rewrite Rules*

*Rewrite rules express basic transformations on terms. A rewrite rule has the* form L : l -> r, where L is the label of the rule, and the term patterns l and r are its left-hand side and right-hand side, respectively. A term pattern is either a variable, a nullary constructor C, or the application C(p1,...,pn)

*module peano signature*

*sorts Nat constructors*

*Zero : Nat*

*Succ : Nat -> Nat*

*Plus : Nat \* Nat -> Nat rules*

*A : Plus(Zero, x) ->* *x*

*B : Plus(Succ(x), y) -> Succ(Plus(x, y))*

*Fig. 1. An example Stratego module with signature and rewrite rules.*

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

*A rule L: l -> r applies to a (ground) term t when the pattern l matches t, i.e., when l has the same top-level structure as t. Applying L to t has the e ect of transforming t to the term obtained by replacing the vari- ables in r with the subterms of t to which they correspond. For exam- ple, rule B transforms the term Plus(Succ(Zero),Succ(Zero)) to the term Succ(Plus(Zero,Succ(Zero))), where x corresponds to Zero and y corre- sponds to Succ(Zero).*

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

*2.2* [*Combining*](#_bookmark4) *Rules with Strategies*

*Figures 2 and 3 illustrate how strategies can be used to control rewriting. Fig-* ure 2 gives a generic de nition of the notion of innermost normalization under some transformation s. The innermost strategy can be instantiated with any selection of rules to achieve normalization of terms under those rules. For in- stance, in Figure 3 the strategy main is de ned to normalize Nat terms using the innermost strategy instantiated with rules A and B. In general, transfor- mation rules and reduction strategies can be de ned independently and can be combined in various ways. A di erent selection of rules can be made, or the rules can be applied using a di erent strategy. Not all rules in a speci - cation are required to participate in a speci c normalization. In this way, it is possible in Stratego to develop a library of valid transformation rules and apply them in various transformations as needed.

*2.3 Rewriting Strategies*

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

*Rewrite rules are just strategies which apply transformations to the roots*

*module innermost strategies*

*innermost(s) = bottomup(red(s)) bottomup(s) = rec r(all(r); s)*

*red(s) = rec x(s; bottomup(x) <+ id)*

*Fig. 2. Generic traversal strategies.*

*module apply-peano imports innermost peano strategies*

*main = innermost(A + B)*

*Fig. 3. Using Peano rules.*

*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 rst attempts to apply s1 to the subject term. If that succeeds, it applies s2 to the result; otherwise it fails. The non-deterministic choice s1 + s2 of strategies s1 and s2 attempts to apply either s1 or s2 to the subject term, but in an unspeci ed order. It succeeds if either s1 or s2 succeeds, and fails otherwise. The deterministic choice s1 <+ s2 of strategies s1 and s2 rst attempts to apply s1 to the subject term. If s1 fails, then it attempts to apply s2 to the subject term. If s1 and s2 both fail, then it fails as well. The recursive closure rec x(s) of a strategy s attempts to apply to the subject term the strategy obtained by replacing each occurrence of the variable x in s by the strategy rec x(s).*

*A strategy de nition f(x1,...,xn) = s introduces a new strategy oper-* ator f parameterized with strategies x1,...,xn and having body s. Such de nitions cannot refer (directly or indirectly) to the operator being de ned. Instead, all recursion must be expressed explicitly by means of the recursion operator rec.

*2.4 Term Traversal*

*The strategy combinators just described combine strategies which apply trans-* formation rules to the roots of their subject terms. In order to apply a rule at an internal site of a term (i.e., to a subterm), it is necessary to traverse the term. Stratego de nes several primitive operators which expose the direct subterms of a constructor application. These can be combined with the op- erators described above to de ne 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 Strat-* ego. For each constructor C there is a corresponding congruence operator

*C. If C is an n-ary constructor, then the corresponding congruence operator de nes the strategy C(s1,...,sn), which applies only to terms of the form C(t1,...,tn) resulting in C(t1',...,tn'), if each si successfully applies to ti resulting in ti'. For example, the congruence Plus(s1,s2) applies only to Plus terms, and it works by applying s1 to the rst summand and s2 to the second, producing Plus(t1',t2'). If the application of si to ti fails for any i, then the application of C(s1,...,sn) to C(t1,...,tn) also fails.*

*The operator all(s) applies s to each of the direct subterms ti of a con-* structor application C(t1,...,tn). It succeeds if and only if all applications to the direct subterms succeed. The resulting term is the constructor applica- tion C(t1',...,tn') where the ti' are the results obtained by applying s to the terms ti. Note that all(s) is the identity on constants, i.e., on construc- tor applications without children. An example of the use of all appears in the strategy bottomup in Figure 2. The strategy expression rec x(all(x); s) speci es that the strategy is rst applied recursively to all direct subterms of a term, and, thereby, to all of its subterms. If that succeeds, then the argu- ment strategy s is applied to the resulting term. This de nition of bottomup captures the generic notion of a post-order traversal over a term.

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

# *3 An Optimized Speci cation of Innermost* [*Reduction*](#_bookmark3)

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

*module apply-peano strategies*

*main =*

*bottomup(rec r(*

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

*+ {x, y: ?Plus(Succ(x), y); <r> Succ(<r> Plus(x, y))}*

*) <+ id))*

*Fig. 4. Optimized strategy.*

*that the rules are tried at each subterm. Since the terms are in normal form,* no actual transformation is done, of course.

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

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

# *4 Derivation*

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

*The goal of the derivation is to fuse the occurrence of bottomup appearing* in the de nition of the strategy red called by innermost with the right-hand sides of the rules A and B. This avoids renormalizing the terms to which vari- ables from the left-hand sides of these rules are bound. To achieve this, we

*rst desugar the rules and then inline (unfold) de nitions in order to arrive*

*at a single expression containing the complete speci cation of the innermost* strategy. The bottomup strategy can then be distributed over the right-hand sides of the rules to which innermost applies; A and B in our running example.

*4.1 Desugaring Rules*

*In Stratego, rules are not primitives. Instead they are expressed in terms* of primitives for matching and building terms. The strategy ?t matches the subject term against the term pattern t. The strategy !t replaces the subject term with the term constructed by instantiating the variables in the term pattern t with their current bindings. The construct {xs:s} delimits the scope of the variables in the strategy s. Thus, a rule L: l -> r is just syntactic sugar for L = {x1,...,xn:?l;!r}, where the xi are the variables occurring in the rule. The example rules from module peano in Figure 1 thus reduce to

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

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

*4.2 Inlining De nitions*

*The rst step of the derivation consists in inlining de nitions, i.e., in replacing* each call to a strategy by the body of its de nition. If f(s1,...,sn) = s is the de nition of strategy operator f, then a call f(s1,...,sn) to that operator can be replaced by s[s1/x1,...,sn/xn], i.e., by the strategy obtained by replacing the formal parameters of the body of f by its actual arguments. In the case of the main strategy in module apply-peano in Figure 3, inlining gives

*(1) innermost(A + B)*

*By the de nition of innermost this expands to*

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

*By the de nition of red, this in turn gives*

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

*Finally, inlining the de nitions of rules A and B gives*

*(4) bottomup(rec r(*

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

*+ {x,y: ?Plus(Succ(x), y); !Succ(Plus(x, y))}*

*); bottomup(r)*

*<+ id))*

*4.3 Sequential Composition over Choice*

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

*(x + y); z -> (x; z) + (y; z)*

*This rule is not valid for all strategy expressions. Consider a term t for which* x and y both succeed, (x;z) fails, and (y;z) succeeds. Then (x + y); z will fail if application of x is attempted. By contrast, (x;z) + (y;z) will always

*succeed since (y;z) does. It is, however, the case that the rule does hold* whenever z is guaranteed to succeed; in this situation, the success or failure of both sides of the rule is determined wholly by the success or failure of *x* and y.

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

*(5) bottomup(rec r(*

*( {x: ?Plus(Zero, x); !x}; bottomup(r)*

*+ {x,y: ?Plus(Succ(x), y); !Succ(Plus(x, y))}; bottomup(r)*

*) <+ id))*

*4.4 Sequential Composition over Scope*

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

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

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

*(6) bottomup(rec r(*

*( {x: ?Plus(Zero, x); !x; bottomup(r)}*

*+ {x,y: ?Plus(Succ(x), y); !Succ(Plus(x, y)); bottomup(r)}*

*) <+ id))*

*4.5 Strategy Application*

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

*(7) bottomup(rec r(*

*( {x: ?Plus(Zero, x); <bottomup(r)> x}*

*+ {x,y: ?Plus(Succ(x), y); <bottomup(r)> Succ(Plus(x, y))}*

*) <+ id))*

*4.6 Distribution of bottomup*

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

*<bottomup(r)> C(t1,...,tn)*

*= {definition of bottomup}*

*<rec x(all(x); r)> C(t1,...,tn)*

*= {recursion}*

*<all(rec x(all(x); r)); r> C(t1,...,tn)*

*= {semantics of sequential composition and all}*

*<r> C(<rec x(all(x); r)>t1,..., <rec x(all(x); r)>tn)*

*= {definition of bottomup}*

*<r> C(<bottomup(r)> t1,...,<bottomup(r)> tn)*

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

*(8) bottomup(rec r(*

*( {x : ?Plus(Zero, x); <bottomup(r)> x}*

*+ {x,y: ?Plus(Succ(x), y);*

*<r> Succ(<r> Plus(<bottomup(r)> x,<bottomup(r)> y))}*

*) <+ id))*

*4.7 Avoiding Renormalization Finally, we use the observation that*

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

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

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

*(9) bottomup(rec r(*

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

*+ {x, y: ?Plus(Succ(x), y); <r> Succ(<r> Plus(x, y))}*

*) <+ id))*

# *5 Implementation*

*In this section we show how the rules used in the derivation in Section* *4*

*can be implemented in Stratego. We start by de ning the abstract syntax* of Stratego programs. We then add overlays to abstract over speci c pat- terns in the abstract syntax that occur often in the rules. Next, we formalize the rules used in the derivation as Stratego rules. Finally, we combine these rules into a strategy that optimizes occurrences of the innermost strategy in Stratego speci cations. The optimization works for all strategies of the form innermost(R1 + ... + Rn) with arbitrary rules Ri.

*5.1 Abstract Syntax*

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

*(10) Scope(["x"],*

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

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

*(11) Scope(["x","y"],*

*Seq(Match(Op("Plus",[Op("Succ",[Var("x")]),Var("y")])),*

*Build(Op("Succ",[Op("Plus",[Var("x"),Var("y")])]))))*

*is the representation for rule B.*

*As suggested by this signature, Stratego supports the built-in data type* String. Syntactic sugar for lists in the form [t1,...,tn] is also provided.

*module stratego*

*signature sorts Term constructors*

*Var*

*Op*

*: String*

*-> Term*

*: String \* List(Term) -> Term*

*sorts SVar Strat SDef*

*constructors* *Id :*

*Fail :*

*Seq : Strat \* Strat Choice : Strat \* Strat LChoice : Strat \* Strat*

*SVar*

*Rec SDef Call All Match Build Scope Where*

*: String*

*: String \* Strat*

*Strat*

*Strat*

*-> Strat*

*-> Strat*

*-> Strat*

*-> SVar*

*-> Strat*

*: String \* List(String) \* Strat -> SDef*

*: SVar \* List(Strat)*

*: Strat*

*: Term*

*: Term*

*: List(String) \* Strat*

*: Strat*

*-> Strat*

*-> Strat*

*-> Strat*

*-> Strat*

*-> Strat*

*-> Strat*

*Fig. 5. Simpli ed abstract syntax of Stratego programs.*

*module strategy-patterns overlays*

*Do(x) = Call(SVar(x),[])*

*Innermost(s, im, r, y) = Bottomup(im, Red(s, r, y)) Bottomup(r, s) = Rec(r, Seq(All(Do(r)), s))*

*Red(s, x, y) = Rec(x, LChoice(Seq(s, Bottomup(y, Do(x))), Id))*

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

*5.2 Patterns in Abstract Syntax*

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

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

*5.3* [*T*](#_bookmark10)*ransformation Rules*

*Figure 7 de nes the rules that were used in the derivation in Section 4. The*

*rst six rules are distribution rules for sequential composition over other op-* erators. The right distribution rules for sequential composition over deter- ministic and non-deterministic choice are parameterized with strategies that decide whether or not the strategy expression to be distributed is guaran- teed to succeed. The AssociateR rule associates composition to the right. The IntroduceApp rule de nes the transformation !t; s -> <s> t. Finally, rule BottomupOverConstructor distributes Bottomup over constructor appli- cation. The rule uses the map strategy operator to distribute the application of Bottomup over the list of arguments of the constructor.

*As Figure 7 suggests, Stratego rules can have conditions which are intro-* duced using the keyword where. Conditional rules apply only if the conditions in their where clauses succeed. In addition, the notation \r\ converts a rule r into a strategy. The argument to map in BottomupOverConstructor is thus the strategy corresponding to the local rule that transforms a term t into an application of the same instance of Bottomup to t.

*module fusion-rules imports stratego rules*

*SeqOverChoiceL :*

*Seq(x, Choice(y, z)) -> Choice(Seq(x, y), Seq(x, z))*

*SeqOverLChoiceL :*

*Seq(x, LChoice(y, z)) -> LChoice(Seq(x, y), Seq(x, z))*

*SeqOverChoiceR(succ) :*

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

*SeqOverLChoiceR(succ) :*

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

*SeqOverScopeR :*

*Seq(Scope(xs, s1), s2) -> Scope(xs, Seq(s1, s2))*

*SeqOverScopeL :*

*Seq(s1, Scope(xs, s2)) -> Scope(xs, Seq(s1, s2))*

*AssociateR :*

*Seq(Seq(x, y), z) -> Seq(x, Seq(y, z))*

*IntroduceApp :*

*Seq(Build(t), s) -> Build(App(s, t))*

*BottomupOverConstructor : App(Bottomup(x, s), Op(c, ts)) ->*

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

*Fig. 7. Distribution and association rules.*

*5.4* [*Str*](#_bookmark11)*ategy*

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

*The fusion strategy sequences four constituent strategies. First, an occur-* rence of the innermost strategy is recognized using the congruence operator corresponding to the Innermost overlay. The IntroduceMark rule applies a mark to the choice of rules to be used in the innermost normalization of terms, and the strategy propagate-mark then propagates the mark to the argument

*module fusion-strategy*

*imports strategy-patterns fusion-rules signature*

*constructors Mark : Strat*

*strategies fusion =*

*Innermost(IntroduceMark,id,?r,id);*

*propagate-mark;*

*fuse-with-bottomup(?Bottomup(\_, Do(r))); alltd(BottomupToVarIsId(?Do(r)))*

*propagate-mark =*

*innermost(SeqOverChoiceL + SeqOverLChoiceL + SeqOverScopeL)*

*fuse-with-bottomup(succ) =* *innermost(SeqOverChoiceR(succ) + SeqOverLChoiceR(succ)*

*+ SeqOverScopeR + AssociateR + IntroduceApp*

*+ BottomupOverConstructor)*

*rules*

*IntroduceMark : s -> Seq(Mark, s) BottomupToVarIsId(isr) :*

*Seq(Mark, Seq(Match(lhs), Build(rhs))) -> Seq(Match(lhs), Build(rhs'))*

*where <tvars> lhs => vs;*

*<alltd(\App(Bottomup(\_,r), Var(v)) -> Var(v)*

*where <fetch(?v)> vs; <isr> r \)> rhs => rhs'*

*Fig. 8. Fusion strategy.*

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

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

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

*where alpha and beta are the abstract syntax representations for rules A and*

*B, respectively, given in (10) and (11), and p, w, and z are new auxiliary* variables corresponding to the bound variables in Figure 2. Applying the speci ed Innermost overlay to (12) yields

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

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

*(14) Innermost(Choice(Seq(Mark,alpha),Seq(Mark,beta)),p,w,z) which corresponds to the strategy in (5).*

*Next, the strategy fuse-with-bottomup distributes trailing occurrences of*

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

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

*(16) Scope(["x"],Seq(Mark,*

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

*and beta1 is*

*(17) Scope(["x","y"],Seq(Mark,*

*Seq(Match(Op("Plus",[Op("Succ",[Var("x")]),Var("y")])), Build(App(Do(w),*

*Op("Succ",*

*[App(Do(w),*

*Op("Plus",*

*[App(Bottomup(z,Do(w)),Var("x")),*

*App(Bottomup(z,Do(w)),Var("y"))]*

*))]))))))*

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

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

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

*It performs all outermost applications of s in the subject term by rst attempt-* ing to apply s to the root of the subject term and, if this fails, recursively

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

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

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

*Applying the alltd traversal speci ed in fusion to (15), for example, gives*

*(18) Bottomup(p,Rec(w,LChoice(Choice(alpha2,beta2),Id))) where alpha2 is*

*(19) Scope(["x"],*

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

*and beta2 is*

*(20) Scope(["x","y"],*

*Seq(Match(Op("Plus",[Op("Succ",[Var("x")]),Var("y")])), Build(App(Do(w),*

*Op("Succ",*

*[App(Do(w),*

*Op("Plus",[Var("x"),Var("y")]))])))))*

*This is the nal result of applying fusion to the abstract syntax representation* of innermost(A+B). It corresponds to the strategy in (9).

# *6 Concluding Remarks*

*In this paper we have shown how local transformations can be used to fuse logic* and control in optimizing abstract programs. Strategies play two important roles in our approach. First, they appear as abstract programming devices that are subject to optimization. Second, together with local transformation rules, they provide a language in which automatic optimizations can be speci ed in an elegant manner. Strategy-based optimization can thus be used to reduce ineÆciencies associated with the genericity of strategies as programming tools. The optimization strategy presented in this paper is included as an experi- mental optimization phase in the Stratego compiler (version 0.5.4). The opti- mization still has some limitations. Only strategies of the form innermost(R1

*+ ... + Rn), where the Ri are rules, are optimized. Strategies such as*

*innermost(rules1 + rules2), where the strategies rules1 and rules2 are* de ned as rules1 = R11 + ... + R1k and rules2 = R21 + ... + R2l , are not handled properly because the inliner currently does not inline such de nitions. This requires a generalization of the inliner. Furthermore, rules with conditions that introduce new variables are handled, but the terms bound to the newly introduced variables are renormalized. Based on an analysis of the conditions this could be avoided under some circumstances.

*Strategies have also been used to optimize programs which are not them-* selves de ned in terms of strategies. In [6], for example, they are used to elimi- nate intermediate data structures from functional programs. In [16], strategies are used to build optimizers for an intermediate format for ML-like programs. In both cases, strategies are used | as they are here | in conjunction with small local transformations to achieve large-scale optimization e ects.

*Small local transformations have been dubbed \humble transformations"* in [14]. Such transformations are used extensively in optimizing compilers based on the compilation-by-transformation idiom [8,9,1,13]. They are also used to some degree in most compilers, although not necessarily recognizable as rewrite rules in the implementation.

*The optimization of innermost presented in this paper was inspired by* more general work on functional program optimization. In [5], 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 semantics in [16].

*The importance of optimizing term traversals in functional transformation* systems is discussed in [10]. Term traversals are modelled there by fold func- tions but, since the fold algebras under consideration are updateable, standard fusion techniques for functional programs [17,11,18] are not immediately ap- plicable. The fusion techniques presented here may nevertheless provide a means of implementing optimizations which automatically shortcut recursion in term traversals. If, as suggested in [10], shortcuts of recursion in term traversals should be regarded as program specialization then, since special- ization can be seen as an automated instance of the traditional fold/unfold program optimization methodology [7], optimization of traversals should in- deed be achievable via fold/unfold transformations. These connections are deserving of further investigation.

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

*Acknowledgments*

*Patricia Johann is supported in part by the National Science Foundation* under grant CCR-9900510. We thank Simon Peyton Jones, Andrew Tolmach, Roy Dyckho 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.*

*6*

*[2] T. Dinesh, M. Haveraaen, and J. Heering. An algebraic programming style for numerical software and its optimization. Scienti c Programming, 8(4), 2001. 1*

*[3] P. Fradet and D. L. M etay*[*er.*](#_bookmark0) *Compilation of functional languages by program transformation. ACM Transactions on Programming Languages and Systems,* *13(1):21{51, January 1991. 1*

[*[4]*](#_bookmark0) *M. Haveraaen, H. A. Friis, and T. A. Johansen. Formal software engineering for computational modeling. Nordic Journal of Computing, 6(3):241{270, 1999.* *1*

*[5] 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. 6*

*[6] P. Johann and E. Visser. Warm* [*fusion*](#_bookmark0)[*in*](#_bookmark12) *Stratego: A case study in the generation of program transformation systems. Annals of Mathematics and Arti cial Intelligence, 29(1{4):1{34, 2000. 1, 6*

*[7] N. Jones, C. Gomard, and P. Sestoft. Partial Evaluation and Automatic Program Generation. Prentice Hall, 1993. 6*

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

*January 1989. 6*

*[9] R. A. Kelsey. Compilation by Program Transformation. PhD thesis, Yale University, May 1989. 6*

*[10] J. Kort, R. Lammel, and J. Visser. F*[*unctional*](#_bookmark12) *Transformation Systems. In Proceedings of the 9th International Workshop on Functional and Logic Programming, Benicassim, Spain, July 2000. 6*

*[11] J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from* [*recursiv*](#_bookmark0)*e de nitions. In S. L. P. Jones, editor, Functional Programming Languages and Computer Architecture (FPCA'95), pages 314{323. ACM Press, June 1995. 1, 6*

*[12] S. S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997. 1*

*[13] 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. 6*

[*[14]*](#_bookmark12) *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. 1,*

*6*

*[15] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A.* [*Middeldorp,*](#_bookmark0) *editor, Rewriting Techniques and Applications (RTA'01), volume 2051 of Lecture Notes in Computer Science, pages 357{361. Springer-Verlag, May 2001. 1*

*[16] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with* [*rewriting*](#_bookmark1) *strategies. ACM SIGPLAN Notices, 34(1):13{26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98). 1, 2, 6*

*[17] P. Wadler. Deforestation: Transforming programs to eliminate trees.*

*Theoretical Computer Science, 73:231{248, 1990. 1, 6*

*[18] H. I. Y. Onoue, Z. Hu and M. Takeichi. A* [*calculational*](#_bookmark12) *fusion system HYLO. In IFIP TC 2 Working Conference on Algorithmic Languages and Calculi, pages 76{106. Chapman and Hall, February 1997. 6*