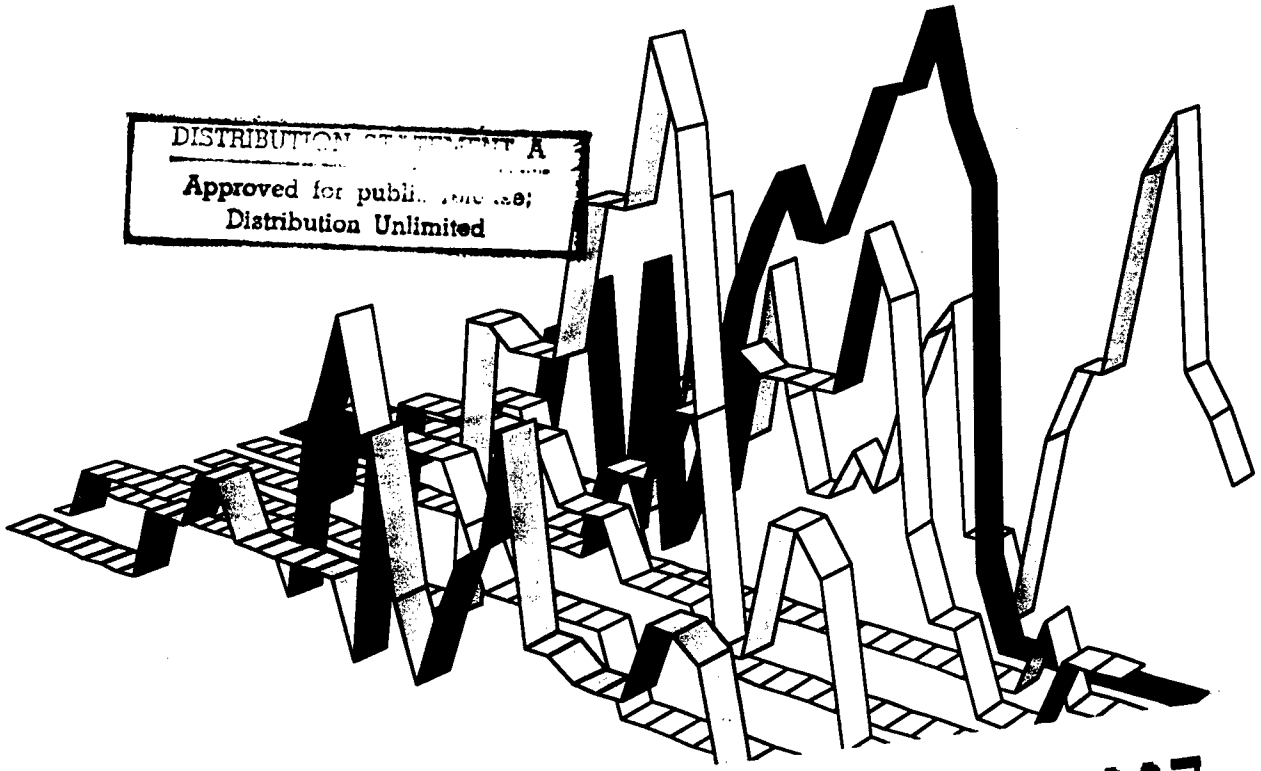
PACIFIC SOFTWARE RESEARCH CENTER
TECHNICAL REPORT

Reduction-Based Optimizer--Initial version

Pacific Software Research Center
April 22, 1998

CONTRACT NO. F19628-96-C-0161
CDRL SEQUENCE NO. [CDRL 0002.5]

Prepared for:
USAF
Electronic Systems Center/AVK



OREGON
GRADUATE
INSTITUTE OF
SCIENCE &
TECHNOLOGY

Reduction-Based Optimizer--Initial version

Pacific Software Research Center
April 22, 1998

CONTRACT NO. F19628-96-C-0161
CDRL SEQUENCE NO. [CDRL 0002.5]

Prepared for:
USAF
Electronic Systems Center/AVK

Prepared for:
Pacific Software Research Center
Oregon Graduate Institute of Science and Technology
PO Box 91000
Portland, OR 97291

Building Program Optimizers with Rewriting Strategies*

Eelco Visser[1], Zine-el-Abidine Benaissa[1], Andrew Tolmach[1,2]
Pacific Software Research Center

[1] Dept. of Comp. Science and Engineering, Oregon Graduate Institute, P.O. Box 91000, Portland, Oregon 97291-1000, USA
[2] Dept. of Computer Science, Portland State University, P.O. Box 751, Portland, Oregon 97207 USA
visser@acm.org, benaissa@cse.ogi.edu, apt@cs.pdx.edu

Abstract

We describe a language for defining term rewriting strategies, and its application to the production of program optimizers. Valid transformations on program terms can be described by a set of rewrite rules; rewriting strategies are used to describe when and how the various rules should be applied in order to obtain the desired optimization effects. Separating rules from strategies in this fashion makes it easier to reason about the behavior of the optimizer as a whole, compared to traditional monolithic optimizer implementations. We illustrate the expressiveness of our language by using it to describe a simple optimizer for an ML-like intermediate representation.

The basic strategy language uses operators such as sequential composition, choice, and recursion to build transformers from a set of labeled unconditional rewrite rules. We also define an extended language in which the side-conditions and contextual rules that arise in realistic optimizer specifications can themselves be expressed as strategy-driven rewrites. We show that the features of the basic and extended languages can be expressed by breaking down the rewrite rules into their primitive building blocks, namely matching and building terms in restricted environments. This primitive representation forms the basis of a simple implementation that generates efficient C code.

1 Introduction

Compiler components such as parsers, pretty-printers and code generators are routinely produced using program generators. The component is specified in a high-level language from which the program generator produces its implementation. Program optimizers are difficult labor-intensive components for which few program generation techniques have been developed to date.

A program optimizer transforms the source code of a program into a program that has the same meaning, but is more efficient. On the level of specification and documentation, optimizers are often presented as a set of correctness-preserving *rewrite rules* that transform code fragments into equivalent more efficient code fragments (e.g., see Table 5). Examples of optimizers for functional programs are discussed in [3, 4, 20]. The paradigm provided by conventional rewrite engines is to compute the normal form of a program with respect to a set of rewrite rules. However, optimizers are usually not implemented in this way. Instead, an algorithm is produced that implements a *strategy* for applying the optimization rules. Such a strategy contains meta-knowledge about the set of rewrite rules and the programming language they are applied to in order to (1) guide the application of rules; (2) guarantee termination of optimization; (3) make optimization more efficient.

Such an ad-hoc implementation of a rewriting system has several drawbacks, even when implemented in a language with good support for pattern matching, such as ML or Haskell. First of all, the transformation rules are embedded in the code of the optimizer making it hard to understand, to maintain, and to reuse individual rules in other transformations. Furthermore, the strategy is not specified at the same level of abstraction as the transformation rules, making it hard to reason about the correctness of the optimizer even if the individual rules are correct.

It would be desirable to apply term rewriting technology directly to produce program optimizers. The standard approach to rewriting is to provide a fixed strategy (e.g. innermost or outermost) for normalizing a term with respect to a set of user-defined rewrite rules. This is not satisfactory when—as is usually the case for optimizers—the rewrite rules are neither confluent nor terminating. A common work-around is to encode a strategy into the rules themselves, e.g., by using an explicit function symbol that controls where rewrites are allowed. But this approach has the same disadvantages as the ad-hoc implementation of rewriting described above: the rules are hard to read, and the strategies are still expressed at a low level of abstraction.

In this paper we argue that a better solution is to use explicit specification of *rewriting strategies*. We show how program optimizers can be built by means of a set of *labeled* rewrite rules and a *user-defined* strategy for applying these rules. In this approach transformation rules can be defined independently of any strategy, so the designer can concentrate on defining a set of correct transformation rules for a programming language. The transformation rules can then be used in many independent strategies that are specified in a formally defined *strategy language*. Given such a high-level specification of a program optimizer, a compiler can generate efficient code for executing the optimization rules.

*This work was supported, in part, by the US Air Force Materiel Command under contract F19628-93-C-0069 and by the National Science Foundation under grant CCR-9503383.

April 15, 1998
Submitted for publication

Starting with simple unconditional rewrite rules as atomic strategies we introduce in Section 2 the basic combinators for building rewriting strategies. We give examples of strategies and define their operational semantics.

In Section 3 we explore optimization rules for RML programs, an intermediate format for ML-like programs [21]. This example shows that there is a gap between the unconditional rewrite rules used in rewriting and the transformation rules used for optimizations. For this reason, we enrich rewrite strategies with features such as conditions, contexts and alpha renaming.

In order to avoid complicating the language by many ad-hoc features, we refine the strategy language in Section 4 by breaking down rewrite rules into the notions of matching and building of terms. In Section 5 we show how this refined language can be used to define rules with conditions and contexts. In Section 6 we use the resulting language to give a formal specification of the RML rules presented earlier.

2 Rewriting Strategies

A rewriting strategy is an algorithm for applying rewrite rules. In this section we introduce the building blocks for specifying such algorithms and give several examples of their application. The strategy language presented in this section is an extension of previous work [16] of one of the present authors.

2.1 Terms

We will represent expressions in the object language by means of first-order terms. A first-order term is a variable, a constant, a tuple of one or more terms, or an application of a constructor to one or more terms. This is summarized by the following grammar:

$$t ::= x \mid c \mid (t_1, \ldots, t_n) \mid f(t_1, \ldots, t_n)$$

where x represents variables (lowercase identifiers), c represents constants (uppercase identifiers or integers) and f represents constructors (uppercase identifiers). We denote the set of all variables by X, the set of terms with variables by $T(X)$ and the set of ground terms (terms without variables) by T. Terms can be typed by means of signatures. For simplicity of presentation, we will consider only untyped terms in this paper until Section 6.

2.2 Rewrite Rules

The basis of a strategy is a set of labeled rewrite rules of the form $\ell : l \to r$, where ℓ is a label, l and r are first-order terms. For example, consider the following rewrite rules on a small language of lists constructed with `Cons` and `Nil` and providing the functions `Conc` and `Rev`.

$$\texttt{Cnc1} : \texttt{Conc}(\texttt{Nil}, xs) \to xs$$
$$\texttt{Cnc2} : \texttt{Conc}(\texttt{Cons}(x, xs), ys) \to \texttt{Cons}(x, \texttt{Conc}(xs, ys))$$
$$\texttt{Rev1} : \texttt{Rev}(\texttt{Nil}, ys) \to ys$$
$$\texttt{Rev2} : \texttt{Rev}(\texttt{Cons}(x, xs), ys) \to \texttt{Rev}(xs, \texttt{Cons}(x, ys))$$

The first two rules define the concatenation of two lists. The last two rules define the reversal of a list by shifting elements of the first list to the second list until the first is empty and the second is the reversed list.

A rewrite rule specifies a single step transformation of a term. For example, rule `Cnc2` induces the following transformation:

```
Conc(Cons(1, Nil), Cons(2, Nil))
--Cnc2--> Cons(1, Conc(Nil, Cons(2, Nil)))
```

In general, a rewrite rule defines a labeled transition relation between terms and reducts, as formalized in the operational semantics in Table 1. A reduct is either a term or $\uparrow$, which denotes failure. The first rule defines that a rule ℓ transforms a term t into a term t' if there exists a substitution σ mapping variables to terms such that t is a σ-instance of the left-hand side l and t' is a σ-instance of the right-hand side r. The second rule states that an attempt to transform a term t with rule ℓ fails, if there is no substitution σ such that t is a σ-instance of l. Note that a rewrite rule applies at the *root* of a term. Later on we will introduce operators for applying a rule to a subterm.

$$t \xrightarrow{\ell:l\to r} t' \quad \text{if } \exists\sigma : \sigma(l) = t \wedge \sigma(r) = t'$$

$$t \xrightarrow{\ell:l\to r} \uparrow \quad \text{if } \neg\exists\sigma : \sigma(l) = t$$

Table 1: Operational semantics for unconditional rules.

2.3 Reduction-Graph Traversal

The reduction graph induced by a set of rewrite rules is the transitive closure of the single step transition relation. It forms the space of all possible transformations that can be performed with those rules.

For instance, one path in the reduction graph induced by the rules `Rev1` and `Rev2` is the following:

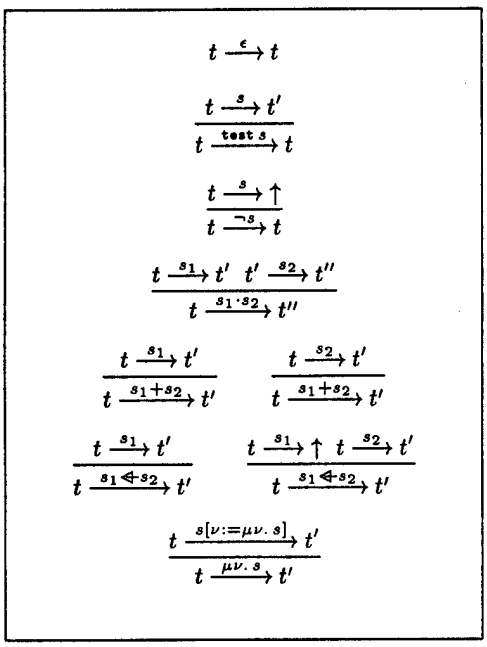
```
Rev(Cons(1, Cons(2, Nil)), Nil)
--Rev2--> Rev(Cons(2, Nil), Cons(1, Nil))
--Rev2--> Rev(Nil, Cons(2, Cons(1, Nil)))
--Rev1--> Cons(2, Cons(1, Nil))
```

A strategy is a compact description of a subset of all such paths. Rewrite rules are atomic strategies that describe a path of length one. In this section we consider combinators for combining rules into more complex strategies. The operational semantics of these strategy operators is defined in Table 2.

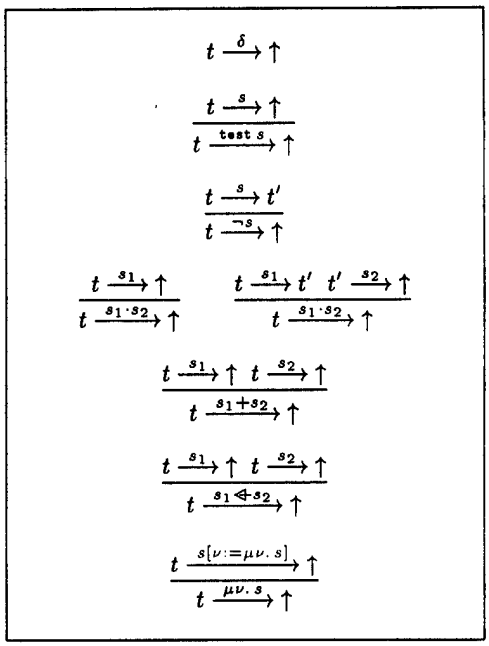
The fundamental operation for compounding the effects of two transformations is the *sequential composition* $s_1 \cdot s_2$ of two strategies[1]. It first applies s_1 and, if that succeeds, it applies s_2. For example, the reduction path above is described by the strategy `Rev2` · `Rev2` · `Rev1`.

The *non-deterministic choice* $s_1 + s_2$ chooses between the strategies s_1 and s_2 such that the strategy chosen succeeds. For instance, the strategy `Rev1` + `Rev2` applies either `Rev1` or `Rev2`. Note that due to this operator there can be more than one way in which a strategy can succeed.

[1]The notation $x \cdot y$ is derived from the process algebra ACP [6] and should not be confused with function composition.



(a) positive rules



(b) negative rules

Table 2: Operational semantics for basic combinators.

Strategies that repeatedly apply some rules can be defined using the *recursion* operator $\mu x.\, s$. One strategy for the complete evaluation of an application of `Rev` is:

$$\mu x.(\texttt{Rev1} + (\texttt{Rev2} \cdot x))$$

It tries to apply either rule `Rev1` or `Rev2`. In the first case (the first argument is `Nil`) evaluation is done. In the second case, the entire strategy is invoked again through the recursion variable x. This strategy will only succeed if it can terminate with an application of `Rev1`.

With the non-deterministic choice operator the programmer has no control over which strategy is chosen. The *deterministic* or *left choice* operator $s_1 \lhd s_2$ is biased to choose its left argument first. It will consider the second strategy only if there is no way in which the first can succeed. This operator can be used to optimize the strategy for evaluating list reversals. The strategy

$$\mu x.((\texttt{Rev2} \cdot x) \lhd \texttt{Rev1})$$

always first tries to apply rule `Rev2` before it considers `Rev1`.

The *identity* strategy ϵ always succeeds. It is often used in conjunction with left choice to build an optional strategy: $s \lhd \epsilon$ tries to apply s, but when that fails just succeeds with ϵ. The *failure* strategy δ is the dual of identity and always fails.

The strategy `test` s can be used to *test* whether a strategy s would succeed or fail without having the transforming effect of s. The *negation* $\neg s$ of a strategy s is similar to test, but tests for failure of s. We will see examples of the application of these operators in Section 6.

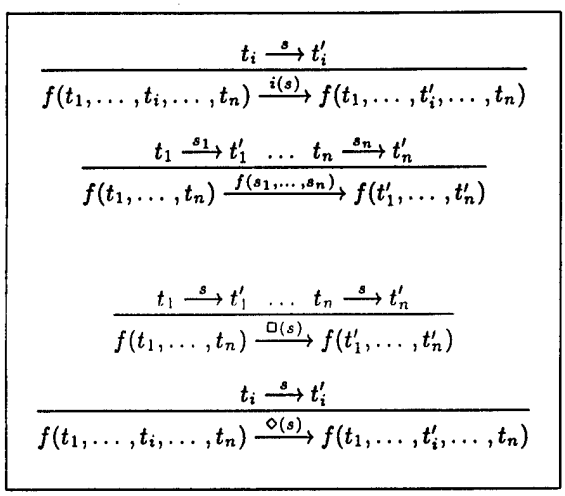
Redex and Normal Form We will call a term an ℓ-redex if it can be transformed with a rule ℓ, otherwise it is in ℓ-normal form. We will generalize this terminology to general strategies, i.e. if $t \xrightarrow{s} t'$, then t is an s-redex and if $t \xrightarrow{s} \uparrow$, then t is in s-normal form.

Strategy Definitions In order to name common patterns of strategies we will use strategy definitions. A definition $f(x_1, \ldots, x_n) = s$ introduces a new n-ary strategy operator f. An application $f(s_1, \ldots, s_n)$ of f to n strategies denotes the instantiation $s[x_1 := s_1 \ldots x_n := s_n]$ of the body of the definition. Strategy definitions are not recursive and not higher-order, i.e. it is not possible to give a strategy operator as argument to a strategy operator. An example of a common pattern is the application of a strategy to a term as often as possible. This is expressed by the definitions

$$\texttt{repeat}(s) = \mu x.((s \cdot x) \lhd \epsilon)$$
$$\texttt{repeat1}(s) = s \cdot \texttt{repeat}(s)$$

The strategy `repeat`(s) applies s zero or more times, but as often as possible. The strategy `repeat1`(s) applies s one or more times, but as often as possible. Using `repeat`, yet another way of evaluating the application of `Rev` is the strategy `repeat(Rev2)` $\cdot$ `Rev1` which is equivalent to $\mu x.((\texttt{Rev2} \cdot x) \lhd \epsilon) \cdot \texttt{Rev1}$.

Backtracking As we remarked before, the non-deterministic choice operator $a + b$ leads to more than one transformation when both strategies s_1 and s_2 are applicable to a term. This leads to the possibility of backtracking. Consider the strategy $(s_1 + s_2) \cdot s_3$. If both s_1 and s_2 apply to a term t, say we have $t \xrightarrow{s_1} t'$ and $t \xrightarrow{s_2} t''$, but s_3 fails for t' and succeeds for t'', i.e. $t' \xrightarrow{s_3} \uparrow$ and $t'' \xrightarrow{s_3} t'''$, then we get as result $t \xrightarrow{(s_1+s_2)\cdot s_3} t'''$. In other words, regardless of the order in which s_1 and s_2 are tried, a succeeding one will be chosen. Or, in more operational terms, if a choice



$$\frac{t_i \xrightarrow{s} t_i'}{f(t_1,\ldots,t_i,\ldots,t_n) \xrightarrow{i(s)} f(t_1,\ldots,t_i',\ldots,t_n)}$$

$$\frac{t_1 \xrightarrow{s_1} t_1' \quad \ldots \quad t_n \xrightarrow{s_n} t_n'}{f(t_1,\ldots,t_n) \xrightarrow{f(s_1,\ldots,s_n)} f(t_1',\ldots,t_n')}$$

$$\frac{t_1 \xrightarrow{s} t_1' \quad \ldots \quad t_n \xrightarrow{s} t_n'}{f(t_1,\ldots,t_n) \xrightarrow{\Box(s)} f(t_1',\ldots,t_n')}$$

$$\frac{t_i \xrightarrow{s} t_i'}{f(t_1,\ldots,t_i,\ldots,t_n) \xrightarrow{\Diamond(s)} f(t_1,\ldots,t_i',\ldots,t_n)}$$

(a) positive rules

$$\frac{t_i \xrightarrow{s} \uparrow}{f(t_1,\ldots,t_i,\ldots,t_n) \xrightarrow{i(s)} \uparrow}$$

$$\frac{t_i \xrightarrow{s_i} \uparrow}{f(t_1,\ldots,t_n) \xrightarrow{f(s_1,\ldots,s_n)} \uparrow}$$

$$g(t_1,\ldots,t_m) \xrightarrow{f(s_1,\ldots,s_n)} \uparrow \quad \text{if } f \neq g$$

$$\frac{t_i \xrightarrow{s} \uparrow}{f(t_1,\ldots,t_n) \xrightarrow{\Box(s)} \uparrow}$$

$$\frac{t_1 \xrightarrow{s} \uparrow \quad \ldots \quad t_n \xrightarrow{s} \uparrow}{f(t_1,\ldots,t_n) \xrightarrow{\Diamond(s)} \uparrow}$$

(b) negative rules

Table 3: Operational semantics for term traversal operators.

made at some point leads to failure later on, the strategy will backtrack to the choicepoint. This does not hold for left choice. Once the left branch has succeeded the right branch can never be chosen. Therefore, left choice provides the means to define deterministic strategies without the global backtracking behaviour described above.

2.4 Term Traversal

The operators we introduced above apply strategies to the root of a term. This is not adequate for achieving all transformations. For instance, for evaluating an application of `Conc` with rules `Cnc1` and `Cnc2`, we need to apply rules to subterms of the root. For example, if we continue the reduction of the concatenation we started before we get the reduction path:

```
    Conc(Cons(1, Nil), Cons(2, Nil))
--Cnc2-->  Cons(1, Conc(Nil, Cons(2, Nil)))
--2(Cnc1)--> Cons(1, Cons(2, Nil))
```

The second step in this reduction is an application of rule `Cnc1` to the second argument of the `Cons`.

In order to apply rewrite rules below the root of a term, i.e. to the subterms of a term, we need operators to traverse the tree structure of a term. For this purpose we introduce four new operators. The operational semantics of these operators is defined in Table 3.

The fundamental operation for term traversal is the application of a strategy to a specific direct subterm of a term. The strategy $i(s)$ applies strategy s to the i-th child. Using this operator an arbitrary path in a term can be constructed. We saw an example above, $2(\texttt{Cnc1})$ applies rule `Cnc1` to the second argument of the root.

The *congruence* operator $f(s_1,\ldots,s_n)$ is a strategy that specifies a strategy to be applied to each direct subterm of a term with constructor f. Instead of $2(\texttt{Cnc1})$ we could use the congruence operator $\texttt{Cons}(\epsilon,\texttt{Cnc1})$ to apply rule `Cnc1` to the second argument of a `Cons` term. Using this idea we can now construct a strategy for evaluating the concatenation of two lists:

$$\mu x.(\texttt{Cnc1} + \texttt{Cnc2} \cdot \texttt{Cons}(\epsilon, x))$$

The strategy repeatedly applies rule `Cnc2` and then terminates with rule `Cnc1`. In the first case the strategy is recursively applied to the `Conc` in the second argument of `Cons`.

A more general example of the use of congruence operators is the strategy $\texttt{map}(s)$ that applies a strategy s to each element of a list:

$$\texttt{map}(s) = \mu x.(\texttt{Nil} + \texttt{Cons}(s, x))$$

The path and congruence operators are useful for constructing strategies for a specific data structure. To construct more general strategies that can abstract from a concrete representation we introduce the operators $\Box(s)$ and $\Diamond(s)$.

The strategy $\Box(s)$ applies s to each direct subterm of the root. This only succeeds if s succeeds for each direct subterm. In case of constants, i.e. constructors without arguments, the strategy always succeeds, since there are no direct subterms. This allows us to define very general traversal strategies. For example, the following strategies apply a strategy s to each node in a term, in preorder (top-down), postorder (bottom-up) and a combination of pre- and postorder (downup):

$$\begin{aligned}\texttt{topdown}(s) &= \mu x.(s \cdot \Box(x))\\ \texttt{bottomup}(s) &= \mu x.(\Box(x) \cdot s)\\ \texttt{downup}(s) &= \mu x.(s \cdot \Box(x) \cdot s)\end{aligned}$$

The strategy $\Diamond(s)$ applies s non-deterministically to one direct subterm. It fails if there is no subterm for which it succeeds. In particular, it fails for constants, since they have no child for which s can succeed. As we did with $\Box$ we can construct bottom-up and top-down traversals with $\Diamond$:

$$\begin{aligned}\texttt{oncetd}(s) &= \mu x.(s \lessdot \Diamond(x))\\ \texttt{oncebu}(s) &= \mu x.(\Diamond(x) \lessdot s)\end{aligned}$$

These strategies succeed if they find an s-redex as subterm.

These strategies perform a fixed traversal over a term. A normalization strategy for a strategy s keeps traversing the term until it finds no more s-redexes. Examples of well-known normalization strategies are `reduce`, which repeatedly finds a redex somewhere in the term, `outermost`, which repeatedly finds a redex starting from the root of the term and `innermost`, which looks for redexes from the leafs of the term. Their definitions are:

$$\begin{aligned}\texttt{reduce}(s) &= \texttt{repeat}(\mu x.(\Diamond(x) + s))\\ \texttt{outermost}(s) &= \texttt{repeat}(\texttt{oncetd}(s))\\ \texttt{innermost}(s) &= \texttt{repeat}(\texttt{oncebu}(s))\end{aligned}$$

Note that this definition of innermost reduction is not very efficient. After finding a redex, search for the next redex starts at the root again. A more efficient definition of innermost reduction is the following.

$$\texttt{innermost}'(s) = \mu x.(\Box(x) \cdot (s \cdot x \lhd \epsilon))$$

It first normalizes all subterms ($\Box(x)$), i.e. all subterms are in s-normal-form. Then it tries to apply s at the root. If that fails this means the term is in s-normal-form and normalization terminates with ϵ. Otherwise, the reduct resulting from applying s is normalized again.

Finally, $\Diamond\!\!\!\!\!\!\Diamond(s)$ is a parallel (greedy) version of $\Diamond(s)$ that is defined by

$$\Diamond\!\!\!\!\!\!\Diamond(s) = \texttt{test}(\Diamond(s)) \cdot \Box(s \lhd \epsilon)$$

The operator is a hybrid between $\Box(s)$ and $\Diamond(s)$. It is like $\Diamond$ because it has to succeed for at least one child and it is like $\Box$ because it applies to all children. The difference with $\Box$ is that it does not have to succeed for all children. An application of $\Diamond\!\!\!\!\!\!\Diamond$ is the strategy

$$\texttt{somebu}(s) = \mu x.((\Diamond\!\!\!\!\!\!\Diamond(x) \cdot (s \lhd \epsilon)) \lhd s)$$

that applies s bottom-up at least once somewhere in the term, but as often as possible.

3 Case Study: RML Optimizer

RML [21] is a strict functional language, essentially similar to the core of Standard ML [18] with a few restrictions. In this paper we consider a subset of RML that includes basic features of functional languages, namely basic constants (integer, boolean, etc.) and primitive built-in functions, tuples and selection, let-bindings and mutually recursive functions. Programs are pre-processed by the compiler of RML to A-normal form. The syntax of this restriction of RML is presented in Table 4.

Table 5 describes a set of meaning preserving source-to-source transformation rules for RML. For in-depth discussions of the intent and correctness of these rules we refer the reader to the literature on transformation of functional programs, e.g. [3, 4, 12, 20]. The rules in Table 5 were inspired by the high-level rules presented in [4]. In the sequel, we concentrate on the details of the implementation of these rules.

It might seem straightforward to implement these rules by a rewriting system using the strategy combinators introduced in the previous section. Unfortunately, this is not the case! There is a gap between these transformation rules and the simple rewrite rules defined above. Only (Hoist1) and (Hoist2) conform to the format. All the other rules use features that are not provided by basic rewrite systems.

$t ::= b \mid t \to t \mid t_1 * \cdots * t_n$	(Types)
$se ::= x \mid c$	(Simple expressions)
$fdec ::= f : t\ x_1, \ldots, x_n = e$	(Function declarations)
$vdec ::= x : t = e$	(Variable bindings)
$e ::= se$	(Expressions)
$\mid x(se_1, \ldots, se_n)$	
$\mid d(se_1, \ldots, se_n)$	
$\mid (se_1, \ldots, se_n)$	
$\mid \text{select}(i, se)$	
$\mid \text{let } vdec \text{ in } e$	
$\mid \text{letrec } fdec_1 \cdots fdec_n \text{ in } e$	

where $x, f, f_1, \ldots$ range over variables, c over constants, and d over primitive built-in functions, i over integers, $e, e_1, \ldots$ over expressions, b over basic types, and $t, t_1, \ldots$ over types.

Table 4: Syntax of RML

(Dead1) and (Dead2) are *conditional* rewrite rules that remove pieces of dead code. The condition (Dead1) tests whether the variable defined by the let occurs in the body of the let. The condition of (Dead2) tests whether any of the functions defined in the *list* of function declarations occurs in the body. (Prop) and (Inline) require *substitution* of free occurrences of a variable by an expression. (Inline) uses *simultaneous substitution* of a list of expressions for a list of variables. In addition, it is a context-sensitive rule, replacing an application of the function f somewhere in the expression e by the body of the function. This is expressed by the use of a *context* $e[f(es)]$. Furthermore, the rule *renames* all occurrences of bound variables with fresh variables, to preserve the invariant that all bound variables are distinct. This invariant simplifies substitution and testing for variable occurrence in an expression. Finally, (Etaexp) generates *fresh variables*, which is a global condition on the whole term.

4 Refining the Strategy Language

The RML example shows that simple unconditional rules lack the expressivity to describe optimization rules for programming languages and that we need enriched rewrite rules with features such as side conditions and contexts and support for alpha renaming and substitution of object variables. For other applications we might need other features such as list matching and matching modulo associativity and commutativity. Adding each of these features as an ad-hoc extension of basic rewrite rules would make the language difficult to implement and maintain. It would be desirable to find a more uniform method to deal with such extensions.

If we take a closer look at the features discussed above, we observe that they all have strategy-like behaviour. For instance, a rule with a context $c[l']$ in the left-hand side and $c[r']$ in the right-hand side can be seen as performing a traversal over the subterm matching c applying rule $l' \to r'$. Therefore, instead of creating more complex primitives such

$$\text{let } v:t = \text{let } vdec \text{ in } e_1 \text{ in } e_2 \longrightarrow \text{let } vdec \text{ in let } v:t = e_1 \text{ in } e_2 \quad \text{(Hoist1)}$$

$$\text{let } v:t = \text{letrec } fdecs \text{ in } e_1 \text{ in } e_2 \longrightarrow \text{letrec } fdecs \text{ in let } v:t = e_1 \text{ in } e_2 \quad \text{(Hoist2)}$$

$$\text{let } v:t = e_1 \text{ in } e_2 \longrightarrow e_2 \quad \text{(Dead1)}$$
$$\text{if } x \notin vars(e_2) \text{ and } e_1 \text{ is pure}$$

$$\text{letrec } fdecs \text{ in } e \longrightarrow e \quad \text{(Dead2)}$$
$$\text{if for all } f:t\,xs = e' \text{ in } fdecs\text{: } f \notin fv(e)$$

$$\text{let } v:t = se \text{ in } e \longrightarrow \text{let } v:t = se \text{ in } e\{se/v\} \quad \text{(Prop)}$$

$$\text{letrec } f:t\,xs = e' \text{ in } e[f(es)] \longrightarrow \text{letrec } f:t\,xs = e' \text{ in } e[rename(e'\{ss/xs\})] \quad \text{(Inline)}$$
$$\text{if } (f \notin ss \cup fv(e[])) \text{ or } (e' \text{ is small})$$

$$\text{let } x:t = (se_1,\cdots,se_n) \text{ in } e[\text{select}(i,x)] \longrightarrow \text{let } x:t = (se_1,\cdots,se_n) \text{ in } e[se_i] \quad \text{(Select)}$$
$$\text{if } f' \text{ and } xs' \text{ are fresh variables}$$

$$\text{let } f:ts \to t = e_1 \text{ in } e_2 \longrightarrow \text{letrec } f:ts \to t\,xs = \text{let } f':ts \to t = e_1 \text{ in } f'\,xs \text{ in } e_2 \quad \text{(Etaexp)}$$
$$\text{if } f' \text{ and } xs \text{ are fresh variables and } e_1 \text{ is pure}$$

Table 5: Transformation rules for RML

as rules with contexts, we break down rewrite rules into their primitives: *matching* against term patterns and *building* terms. Using these primitives we can implement a wide range of features in the strategy language itself by translating rules which use those features to strategy expressions.

Match, Build and Scope We first need to define the semantics of matching and building terms. A rewrite rule $\ell : l \to r$ first matches the term against the left-hand side l with as result a binding of subterms to the variables in l. Subsequently it builds a new term by instantiating the right-hand side r with those variable bindings. By introducing the new strategy primitives `match` and `build` we can break down ℓ into a strategy $\texttt{match}(l) \cdot \texttt{build}(r)$. However, this requires that we carry the bindings obtained by `match` over the sequential composition to `build`. For this reason, we introduce the notion of environments explicitly in the semantics.

An environment $\mathcal{E}$ is a mapping of variables to ground terms. We denote the instantiation of a term t by an environment $\mathcal{E}$ by $\mathcal{E}(t)$. An environment $\mathcal{E}'$ is an extension of environment $\mathcal{E}$ (notation $\mathcal{E}' \sqsupseteq \mathcal{E}$) if for each $x \in \mathsf{dom}(\mathcal{E})$ we have $\mathcal{E}'(x) = \mathcal{E}(x)$. An environment $\mathcal{E}'$ is the smallest extension of $\mathcal{E}$ with respect to a term t (notation $\mathcal{E}' \sqsupseteq_t \mathcal{E}$), if $\mathcal{E}' \sqsupseteq \mathcal{E}$ and if $\mathsf{dom}(\mathcal{E}') = \mathsf{dom}(\mathcal{E}) \cup \mathsf{vars}(t)$.

Now we can formally define the semantics of `match` and `build`. We extend the reduction relation $\xrightarrow{s}$ from a relation between terms and reducts to a relation on pairs of terms and environments, i.e. a strategy s transforms a term t and an environment $\mathcal{E}$ into a transformed term t' and an extended environment $\mathcal{E}'$, denoted by $t : \mathcal{E} \xrightarrow{s} t' : \mathcal{E}'$, or fails, denoted by $t : \mathcal{E} \xrightarrow{s} \uparrow$. The operational semantics of the environment operators is defined in Table 6.

Once a variable is bound it cannot be rebound to a different term. To use a rule more than once we introduce variable scopes. A scope $\{\vec{x} : s\}$ locally undefines the variables $\vec{x}$. The notation $\mathcal{E}/\vec{x}$ denotes $\mathcal{E}$ without bindings for variables in $\vec{x}$. $\mathcal{E}|\vec{x}$ denotes $\mathcal{E}$ restricted to $\vec{x}$.

We have changed the format of the operational semantics. Therefore, we should change all rules in Tables 2 and 3 as follows: replace each $t \xrightarrow{s} t'$ by $t : \mathcal{E} \xrightarrow{s} t' : \mathcal{E}'$

5 Implementation of Transformation Rules

We now have a strategy language that consists of `match` and `build` as atomic strategies (instead of rewrite rules) and all the combinators introduced in Section 2. Using this refined strategy language, we can implement transformation rules by translating them to strategy expressions. In this higher-level view of strategies we can use both the 'low-level' features match, build and scope and the 'high-level' features such as contexts and conditions. We start by defining the meaning of unconditional rewrite rules in terms of our refined strategy language.

5.1 Unconditional Rewrite Rules Revisited

A labeled rewrite rule $\ell : l \to r$ translates to a strategy definition

$$\ell = \{\mathsf{vars}(l,r) : \texttt{match}(l) \cdot \texttt{build}(r)\}$$

It introduces a local scope for the variables used in the rule $\mathsf{vars}(l,r)$, matches the term against l and then builds r using the binding obtained by matching.

5.2 Subcomputation

Many transformation rules require a subcomputation in order to achieve the transformation from left-hand side to right-hand side. For instance, the inlining rule in Table 5 applies a substitution and a renaming to an expression in the right-hand side.

Where The `where` clause is the basic extension to rewrite rules to achieve subcomputations. A rule

$$\ell : l \to r \texttt{ where } s$$

corresponds to the strategy

$$\ell = \{\mathsf{vars}(l,r,s) : \texttt{match}(l) \cdot s \cdot \texttt{build}(r)\}$$

$$t : \mathcal{E} \xrightarrow{\texttt{match}(t')} t : \mathcal{E}' \quad \text{if } \mathcal{E}' \sqsupseteq_{t'} \mathcal{E} \wedge \mathcal{E}'(t') = t$$

$$t : \mathcal{E} \xrightarrow{\texttt{build}(t')} \mathcal{E}(t') : \mathcal{E} \quad \text{if } \mathsf{vars}(t') \subseteq \mathsf{dom}(\mathcal{E})$$

$$\frac{t : \mathcal{E}/\vec{x} \xrightarrow{s} t' : \mathcal{E}'}{t : \mathcal{E} \xrightarrow{\{\vec{x}:s\}} t' : \mathcal{E}'/\vec{x} \cup \mathcal{E}|\vec{x}}$$

(a) positive rules

$$t : \mathcal{E} \xrightarrow{\texttt{match}(t')} \uparrow \quad \text{if } \neg\exists \mathcal{E}' \sqsupseteq_{t'} \mathcal{E} \wedge \mathcal{E}'(t') = t$$

$$t : \mathcal{E} \xrightarrow{\texttt{build}(t')} \uparrow \quad \text{if } \mathsf{vars}(t') \not\subseteq \mathsf{dom}(\mathcal{E})$$

$$\frac{t : \mathcal{E}/\vec{x} \xrightarrow{s} \uparrow}{t : \mathcal{E} \xrightarrow{\{\vec{x}:s\}} \uparrow}$$

(b) negative rules

Table 6: Operational semantics for environment

that first matches l, then executes s and finally builds r. The strategy s can be any strategy that affects the environment in order to bind variables used in r. Note that s can transform the original term, but the effect of this is canceled by the subsequent build. Only the side-effect of s on the environment matters.

Matching Condition A frequently occuring subcomputation is to apply a strategy to a term built with variables from the left-hand side l and match the result against a pattern with variables used in the right-hand side r. The notation

$$\langle s \rangle\ t \Rightarrow t'$$

corresponds to the strategy

$$\texttt{build}(t) \cdot s \cdot \texttt{match}(t')$$

It first evaluates t with respect to s then matches the result (if it succeeded) against t' with as result a side-effect on the variables in t'. If the match to t' is not needed, then $\langle s \rangle\ t$ can be used either to get the side-effect of s or to only test for success of s.

Application in Right-hand Side Often it is annoying to introduce an intermediate name for the result of applying a strategy to a subterm of the right-hand side. Therefore, the application $\langle s \rangle\ t$ can be used directly in the right-hand side r. That is, a rule

$$\ell : l \rightarrow r\{\langle s \rangle\ t\}$$

is an abbreviation of

$$\ell : l \rightarrow r\{x\} \texttt{ where } \langle s \rangle\ t \Rightarrow x$$

where x is a new variable and $r\{\langle s \rangle t\}$ denotes a meta-context, i.e. a term with an occurrence of $\langle s \rangle\ t$.

Conditions Conditions that check whether some predicate holds can also be be seen as subcomputations. We implement these conditions as strategies using the **where** clause. Failure of such a strategy means that the condition does not hold, while a success means that it does hold. Predicates are user-defined strategy operators. For instance, to test that t_1 is a subterm of t_2 the condition $\langle \texttt{in} \rangle\ (t_1, t_2)$ can be used. The predicate `in` is defined as

$$\texttt{in} = \{t_1, t_2 : \texttt{match}((t_1, t_2)) \cdot (\texttt{oncetd}(\texttt{match}(t_1)))\ t_2\}$$

Conditions can be combined by means of the strategy combinators. In particular, conjunction of conditions is expressed by means of sequential composition and disjunction by means of non-deterministic choice.

5.3 Contexts

A useful class of rules are those whose left-hand sides do not match a fixed pattern but match a top pattern and some inner patterns which occur in *contexts*. For instance, consider the (Inline) and (Select) rules in Table 5. Contexts can also be implemented with the **where** clause. A rule

$$\ell : l\{c[l']\} \rightarrow r\{c[r']\}$$

with one context $c[]$ occurring in the left-hand side and right-hand side corresponds to the rule

$$\ell : l\{x\} \rightarrow r\{x'\}$$
$$\texttt{where}(\texttt{oncetd}(\{\mathsf{vars}(l', r')/\ \mathsf{vars}(l, r) : \langle l' \rightarrow r' \rangle\}))\ x \Rightarrow x'$$

where x and x' are fresh variables. The notation $\langle l' \rightarrow r' \rangle$ is an abbreviation for $\texttt{match}(l') \cdot \texttt{build}(r')$ and is used to inline a rule in a strategy. The strategy in the where clause traverses the subterm matching the x (using `oncetd`) to find one occurrence of l' and replaces it with r'. The result of the traversal is assigned to x', which is then used in the right-hand side of the rule. Note that we scope locally the variables of l' and r' except those common to the variables of l and r, since they are bound by the matching of l.

The implementation of the rule above replaces exactly one occurrence of l' in the redex due to the strategy `oncetd`. To replace all occurrences of l' in the context, we have defined a greedy context, written $e[|t|]$. The implementation of this context is the same as the contexts above, except that the traversal strategy `sometd` is used instead of `oncetd`.

5.4 Alpha Renaming

An important feature of program manipulation is bound variable renaming. A major requirement is to provide renaming as an object language independent operation. This means that the designer should indicate the binding constructs of the language. This is done by mapping each binding construct to the list of variables that it binds. For example, for the `Let` construct, the rule

$$\texttt{Bind1} : \texttt{Let}(\texttt{Vdec}(t, v, e), e') \rightarrow [v];$$

gives the binding variable v (see Appendix A for the other rules). Given these rules the strategy `rename` renames all

bound object variables in the term to which it is applied. It is defined using the strategy language (see the definition in Appendix D). This strategy uses the built-in strategy `new` which generates *fresh names*.

6 Rules and Strategy for RML

Rules Table 7 presents the specification of RML optimization. It consists of a signature, rewrite rules and strategy definitions. The signature allows us to statically check the rules, strategies and input term. Because the input of the optimizer are programs in abstract syntax we use the abstract syntax of RML programs instead of concrete syntax.

One benefit of rewrite strategies is that the specification of RML is almost similar to the high-level rules presented in Table 5. There are very few changes, namely the use of greedy context for efficiency considerations, and the bind rules in Appendix A.

Strategies Another important advantage of our approach is the ability to experiment and reason easily with strategies, which are generally *heuristic*. We present two possible strategies: `optimize1` and `optimize2`. No change of rules is required. A separation of strategies from rules prevents many mistakes and enables us to reason on their property such as termination. For instance in `optimize1` and `optimize2`, we have avoided to apply `EtaExp` repeatedly since this rule is not terminating. Both `optimize1` and `optimize2` first apply `EtaExp` once everywhere in the term. The strategy `optimize1` uses the generic strategies `innermost'` and `somedownup` (see Appendix B) to apply the rest of these rules. The strategy `somedownup` is a variant of `sometd` that applies a strategy `s` at all positions of a term. It fails when none of these applications succeed. If it it succeeds we know that some redex has been reduced. Hence, we can repeat `oncedownup` to normalize a term.

While `optimize1` uses generic strategies, `optimize2` performs specific analyses to apply rules. It first tries to hoist a `Let` at the root. Notice that it repeats `Hoist1` since it may reapply at the root, whereas `Hoist2` cannot reapply after one application. Then, only `Let` or `Letrec` expressions can be redexes. For each case there are specific rules that can apply. This leads us to define a sub-strategy for each case and compose them non-deterministically. In both cases we first normalize the body of the `Let` or `Letrec` expression. For a `Let` we try the rules `Prop` and `Sel` and then `Dead1`. For a `Letrec`, we first normalize the bodies of the functions of the `Letrec` expression. Then we try `Inl1` or `Inl2` and if they succeed we try `Dead2`. Since inlining gives rise to new opportunities for optimization, we retry to strategy to this term.

7 Implementation

The strategy language presented in this paper has been implemented in SML. The programming environment consists of a simple interactive shell that can be used to load specifications and terms, to apply strategies to terms using an interpreter and to inspect the result. A simple inclusion mechanism is provided for modularizing specifications. The current implementation does not yet implement the sort checking for rules and strategies. In addition to an interpreter the environment contains a compiler. It compiles a strategy to a C program that transforms terms according to the strategy.

The compilation of non-deterministic strategies is reminiscent of the implementation of Prolog in WAM [1] using success and failure continuations and a stack of choicepoints to implement full backtracking. A difference with WAM is that our implementation deals with choicepoints occuring inside a traversal as in the strategy $\Box(s_1 + s_2) \cdot s_3$.

The run-time environment of compiled strategies is based on the ATerm C-library [19]. It provides functionality for building and manipulating a term data-structure, reference count garbage collection, a parser and pretty-printer for terms. An important feature is that full sharing of terms is maintained (hash-consing) to reduce memory usage.

We have used the implementation to experiment with the optimizer for RML discussed in this paper, but more work is needed before we can present performance results. The strategy language provides many opportunities for optimization. We plan to apply our technique to optimizing strategies.

8 Related Work

Program Optimization There have been many attempts to build frameworks for program analysis and optimization, often using special-purpose formalisms. Systems close to ours in spirit include TXL [9, 17], Puma [13], OPTIMIX [5], and KHEPERA [11]. All these systems provide tree transformation languages with succinct primitives for matching subtrees. Most of these languages require tree traversal to be programmed explicitly. TXL includes a "searching" version of the match operator which behaves like an application of our `topdown` strategy. KHEPERA provides a built-in construct to iterate over the immediate children of a node.

Other recently-proposed optimization frameworks tend to rely on general-purpose languages to describe transformations. Aspect-Oriented Programming [14] advocates the use of domain-specific "aspect" languages to describe optimization of program IR trees; however, existing examples appear to use LISP for this purpose. Intentional Programming [2] provides a library of routines for manipulating ASTs; in principle, these routines can be invoked from a variety of (intentional representations of) languages, but the current implementation uses C-style programs.

Strategies First-order algebraic specification formalisms such as ASF+SDF [10] provide a fixed strategy for normalizing terms with respect to a set of rewrite rules. A common work-around to implement strategies in such a setting is to encode a strategy into the rewrite system by providing an extra outermost constructor that determines at which point in the term a rewrite rule can be applied.

Originating in theorem proving tactics, rewriting strategies were introduced in the algebraic specification languages ELAN [7] and Maude [8]. Maude is a specification formalism based on rewriting logic. It provides equations that are interpreted with innermost rewriting and labeled rules that are used with an outermost strategy. Strategies for applying labeled rules can be defined in Maude itself by means of reflection.

ELAN provides a built-in strategy language similar to the one in this paper. The strategy language described in this paper is a generalization of the language of ELAN. The

```
imports lib list subs props
signature
  sorts TExp Vdec Fdec Se Exp
  operations
    Funtype    : List(TExp) * TExp          -> TExp  -- Type expressions
    Recordtype : List(TExp)                 -> TExp
    Primtype   : String                     -> TExp
    Vdec       : TExp * String * Exp        -> Vdec  -- Variable declarations
    Fdec       : TExp * String * List(String) * Exp -> Fdec  -- Function declarations
    Const      : TExp * String              -> Se    -- Simple expressions
    Var        : String                     -> Se
    Simple     : Se                         -> Exp   -- Expressions
    Record     : List(Se)                   -> Exp
    Select     : Int * Se                   -> Exp
    Papp       : String * List(Se)          -> Exp
    App        : Se * List(Se)              -> Exp
    Let        : Vdec * Exp                 -> Exp
    Letrec     : List(Fdec) * Exp           -> Exp
rules

Hoist1 : Let(Vdec(t, v, Let(vdec, e1)), e2) -> Let(vdec, Let(Vdec(t, v, e1), e2));

Hoist2 : Let(Vdec(t, v, Letrec(fdecs, e1)), e2) -> Letrec(fdecs, Let(Vdec(t, v, e1), e2));

Prop   : Let(Vdec(t, v, Simple(s)), e[| Var(v) |]) -> Let(Vdec(t, v, Simple(s)), e[| s |]);

Dead1  : Let(Vdec(t, v, e1), e2) -> e2 where <not(in)> (v, e2) . <pure> e1;

Dead2  : Letrec(fdecs, e1) -> e1 where <map({f : match(Fdec(_, f, _, _)). <not(in)> (f, e1)})> fdecs;

Inl1   : Letrec([Fdec(t, f, xs, e1)], e2[| App(Var(f), ss) |]) ->
         Letrec([Fdec(t, f, xs, e1)], e2[| <subs . rename> (xs, ss, e1) |]) where <small> e1;

Inl2   : Letrec([Fdec(t, f, xs, e1)], e2[ App(Var(f), ss)              ]) ->
         Letrec([Fdec(t, f, xs, e1)], e2[ <subs . rename> (xs, ss, e1) ])
         where <not(in)> (Var(f), e1) . <not(in)> (Var(f), e2[Hole]);

Sel    : Let(Vdec(t, v, Record(ss)), e[| Select(i, Simple(Var(v))) |]) ->
         Let(Vdec(t, v, Record(ss)), e[| <index> (i, ss) |]);

EtaExp : Let(Vdec(Funtype(ts, t), f1, e1), e2) -> Letrec([Fdec(Funtype(ts, t), f1, xs,
                                        Let(Vdec(Funtype(ts, t), f2, e1), App(Var(f2), xs)))], e2)
         where <pure> e1 . <new> f1 => f2 . <map(new . {x: <x -> Var(x)>})> ts => xs
strategies

group1 = Inl1 + Inl2 + Sel + Prop
group2 = (Dead1 + Dead2) <+ (Hoist1 + Hoist2)

opt1 = innermost'(Hoist1 + Hoist2) .
       somedownup((group1 . repeat(Dead1 + Dead2) <+ repeat1(Dead1 + Dead2)))
optimize1 = bottomup(try(EtaExp)) . repeat(opt1)

opt2 = rec x . (repeat(Hoist1) . try(Hoist2) .
                try( Let(id, x) . try(Prop + Sel) . try(Dead1)
                   + Letrec(id, x) . (Dead2 <+ try(Letrec(map(Fdec(id,id,id,x)),id) .
                                                   try((Inl1 + Inl2) . try(Dead2) . x))))
optimize2 = bottomup(try(EtaExp)) . opt3
```

Table 7: Specification of RML transformation rules

resulting language is a combination of ideas from the process algebra ACP [6] and the modal mu-calculus [15]. An earlier version or our language was described in [16]. Technical contributions of our strategy language include the modal operators $\Box$, $\Diamond$ and $\Diamond$ that enable very concise specification of term traversal; the explicit recursion operator $\mu x.\, s$; the refinement of rewrite rules into `match` and `build`; and the encoding of complex rewriting features into strategies, in particular the expression of rules with contexts.

9 Conclusions

We have illustrated how separating transformation rules from the application strategy can promote concise, understandable descriptions of complex rewriting tasks. Our example compiler optimizer takes about 50 lines; the corresponding handwritten Standard ML code is several hundred lines. Moreover, we can completely alter the optimizer's rewriting strategy by changing just two or three lines; similar changes to the ML version would require extensive structural edits throughout the code.

Although we concentrate on program optimizers in this paper, we believe that the techniques are equally well applicable in other areas where source to source transformations are used, including simplification, typechecking, interpretation and software renovation.

Acknowledgements We thank Bas Luttik for the discussions that started our work on strategies. Several ideas that didn't make it into [16] have been included here. The implementation of the strategy language was speeded up considerably by the use of Tim Sheard's programs for generation of C code and by the use of Pieter Olivier's ATerm library.

References

[1] Hassan Aït-Kaci. *Warren's Abstract Machine. A Tutorial Reconstruction.* The MIT Press, Cambridge, Massachusetts, 1991.

[2] William Aitken. Brian Dickens. Paul Kwiatkowski, Oege de Moor, David Richter, and Charles Simonyi. Transformation in intentional programming. In *Proc. ICRS5*, June 1998. (To appear.).

[3] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[4] Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5):515–540, September 1997.

[5] Uwe Assmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In *Proc. Compiler Construction 1996*, number 1060 in Lecture Notes in Computer Science, 1996.

[6] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information & Control*, 60:82–95, 1984.

[7] Peter Borovanský, Claude Kirchner, and Héléne Kirchner. Controlling rewriting by rewriting. *Electronic Notes in Theoretical Computer Science*, 4, September 1996. In J. Meseguer (ed.) *First International Workshop on Rewriting Logic and its Applications.* Asilomar Conference Center, Pacific Grove, CA, September 3-6, 1996.

[8] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of maude. In José Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89. Elsevier, 1996.

[9] James R. Cordy, Ian H. Carmichael, and Russell Halliday. *The TXL Programming Language, Version 8*, April 1995.

[10] A. Van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific, Singapore, September 1996.

[11] Rickard E. Faith, Lars S. Nyland, and Jan F. Prins. KHEPERA: A system for rapid implementation of domain specific languages. In *Proc. USENIX Conference on Domain-Specific Languages*, pages 243–255, October 1997.

[12] Pascal Fradet and Daniel Le Métayer. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems*, 13(1):21–51, January 1991.

[13] Joseph Grosch. Puma - a generator for the transformation of attributed trees. Technical Report 26, Gesellschaft für Mathematik und Datenverarbeitung mbH, Forschungsstelle an der Universität Karlsruhe, November 1991.

[14] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Technical report, Xerox Palo Alto Research Center, 1997.

[15] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, pages 333–354, 1983.

[16] Bas Luttik and Eelco Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag.

[17] Andrew Malton. The denotational semantics of a functional tree-manipulation language. *Computer Languages*, 19(3):157–168, 1993.

[18] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT press, 1997.

[19] Pieter Olivier. *Term data-structure library — C API.* Programming Research Group, University of Amsterdam, 1997. (Unpublished software documentation.).

[20] S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 1998. (To appear.).

[21] Andrew Tolmach and Dino Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 1998. to appear.

A User-defined RML Predicates

```
(* file: props.r
   --- Some properties of RML expressions *)

rules

Bind1 : Let(Vdec(t, v, e), e') -> [v];
Bind2 : Letrec(fdecs, e) ->
        <map({f: <Fdec(_, f, _, _) -> f>}) > fdecs
Bind3 : Fdec(_,_,xs,_) -> xs

strategies

rmlrename = rename(Bind1 + Bind2 + Bind3)

small = Simple(id) + Record(id) + Select(id, id) +
        Papp(id, id) + App(id, id)

pure = not(oncebu(match(Papp("assign", _))))
```

B Generic Strategies

In this and the next appendices we present three sets of generally applicable strategy operators. Note that all, one, and some stands for $\Box$, $\Diamond$, and $\circledcirc$.

```
(* file: lib.r --- Standard strategies *)
strategies

(* Try s *)

try(s) = s <+ id

(* Repetition *)

repeat(s)  = rec x . ((s . x) <+ id)
repeat1(s) = s . repeat(s)

(* Traversal; all s applications have
   to succeed *)

bottomup(s) = rec x . (all(x) . s)
topdown(s)  = rec x . (s . all(x))
downup(s)   = rec x . (s . all(x) . s)

downup2(s1, s2) = rec x . (s1 . all(x) . s2)

(* Traversal; one s application
   has to succeed *)

oncebu(s) = rec x . (one(x) <+ s)
oncetd(s) = rec x . (s <+ one(x))

(* Greedy traversal; apply s as often as possible
   and at least once. *)

somebu(s) = rec x . (some(x) <+ s)
sometd(s) = rec x . (s . all(s <+id) <+ some(x))

(* Greedier *)

somedownup(s) = rec x . ((s . (all(x) . (s <+ id)
                <+ id)) <+ (some(x) . (s <+ id)))

(* Normalization strategies *)

reduce(s)     = repeat(rec x . (some(x) + s))
outermost(s)  = repeat(oncetd(s))
innermost(s)  = repeat(oncebu(s))
innermost'(s) = rec x . (all(x) . (s . x <+ id))
```

C Lists and Pairs

Lists are constructed with the polymorphic constructors Cons and Nil. Finite lists can be constructed with the special notation $[t_1, \ldots, t_n]$, abbreviating $\texttt{Cons}(t_1, \ldots \texttt{Cons}(t_n, \texttt{Nil}))$. Lists have type $\texttt{List}(A)$ with A some type. Tuples $(t_1, \ldots, t_n)$ have type $\texttt{Prod}([A_1, \ldots, A_n])$, where A_i is the type of t_i.

```
(* file: list.r *)
signature
  operations
    Zip : Prod([List(A), List(B)])
          -> List(Prod([A, B]))
rules

Hd   : Cons(x,l) -> x;
Tl   : Cons(x,l) -> l;
Fst  : (x, y)    -> x;
Snd  : (x, y)    -> y;

Zip1 : Zip(Nil, Nil) -> Nil;
Zip2 : Zip(Cons(x, xs), Cons(y, ys)) ->
       Cons((x, y), Zip(xs, ys));

Ind1 : (1, Cons(x, xs)) -> x;
Ind2 : (n, Cons(x, xs)) -> (n-1, xs) where geq(n,2)

strategies

(* Evaluation strategies *)

zip(s) = rec x . (Zip1 + Zip2 . Cons(s, x))
index  = repeat(Ind2) . Ind1

(* Concatenation *)

conc = {l: match((l, _)) . Snd .
           rec x. (Cons(id, x) <+ build(l))}

(* Find first list element for which s succeeds *)

fetch(s) = rec x . (Cons(s, id) <+ Cons(id, x))

(* Apply strategy to each element of a list *)

map(s) = rec x . (Nil + Cons(s, x))
```

D Substitution and Renaming

```
(* file: subs.r *)

strategies

(* Test occurrence of a in b *)

in = {a: match((a, _)) . Snd . oncebu(match(a))}
```

```
(* Substitution *)

subs = {lst, xs, ss, t:
        match((xs, ss, t)) .
        <zip(id)> Zip(xs, ss) => lst .
        <topdown(<Var(x) -> z
         where <fetch(match(x, z))> lst> <+ id)> t}

(* Renaming *)

rules

Init  : t -> (t, [])

Fresh : x -> (x, <new> x)

Ren   : (x, l) -> z where <fetch(match((x, z)))> l

strategies

binds(s) = {t, l: <(t, l) ->
                   (t, <conc> (<s . map(Fresh)> t, l))>}

dist(s) = {l, t: <(t, l) -> t> .
                 all({x : <x -> (x, l)>} . s)}
rename(s) = Init .
            rec x . (Ren <+ ((binds(s) <+ id) . dist(x)))
```