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 variable binding environments. This gives us a low-level core language which has a clear semantics, can be implemented straightforwardly and can itself be optimized. The current implementation generates C code from a strategy specification.

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 that are usually still developed manually, despite many attempts at producing optimizer generators (e.g., [19, 12, 28, 25, 18, 11]).

*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.

To appear in P. Hudak and C. Queinnec (editors). The 1998 International Conference on Functional Programming (ICFP'98). ACM, Baltimore, MD, September, 1998.

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). This is particularly attractive for functional language compilers (e.g., [3, 4, 24]) that operate via successive small transformations, and don't rely on analyses requiring significant auxiliary data structures. 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) control 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 them hard to understand, to maintain, and to reuse individual rules in other transformations. Secondly, 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. Finally, the host language has no awareness of the transformation domain underlying the implementation and can therefore not use this domain knowledge to optimize the optimizer itself.

It would be desirable to apply term rewriting technology directly to produce program optimizers. However, 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.

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 [26]. This example shows that there is a gap between the unconditional rewrite rules used in rewriting and the transformation rules used for optimization. For this reason, we need to enrich rewrite rules with features such as conditions and contexts. In order to avoid complicating the implementation by many ad-hoc features, we refine our language by breaking down rewrite rules into the notions of matching and building terms (Section 4). This gives us a low-level core language which has a clear semantics, can be implemented straightforwardly and can itself be optimized. The current implementation generates C code from a strategy specification. In Section 5 we show how this core language can be used to encode high-level rules with conditions and contexts. In Section 6 we use the resulting language to give a formal specification of the RML rules presented earlier. Section 7 describes the implementation and Section 8 discusses related work.

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 [20] 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. For now, we assume that a signature Σ is a function mapping operators to their arities. We will also use a shorthand notation for lists. A term $[t_1, t_2, \ldots, t_n]$ denotes a term

$$\mathtt{Cons}(t_1, \mathtt{Cons}(t_2, \ldots, \mathtt{Cons}(t_n, \mathtt{Nil})))$$

Constants are considered to be constructors with zero-arity and tuples are considered to be constructed with a special constructor for each arity.

2.2 Rewrite Rules

A labeled rewrite rule has the form $\ell : l \to r$, where ℓ is a label and l, r are first-order terms. For example, consider a calculus of lists constructed with `Cons` and `Nil` and Boolean values `True` and `False` that defines transformation rules for the constructor `Member` as:

$$\begin{aligned}
&\mathtt{Mem1} : \mathtt{Member}(x, \mathtt{Nil}) \to \mathtt{False}\\
&\mathtt{Mem2} : \mathtt{Member}(x, \mathtt{Cons}(x, ys)) \to \mathtt{True}\\
&\mathtt{Mem3} : \mathtt{Member}(x, \mathtt{Cons}(y, ys)) \to \mathtt{Member}(x, ys)
\end{aligned}$$

A rewrite rule specifies a single step transformation of a term. For example, rule `Mem3` induces the following transformation:

$$\begin{aligned}
&\mathtt{Member(A,\ Cons(B,\ Cons(A,\ Nil)))}\\
\xrightarrow{\mathtt{Mem3}}\ &\mathtt{Member(A,\ Cons(A,\ Nil))}
\end{aligned}$$

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. For instance, in our membership example we have

$$\mathtt{Member(A,\ Cons(B,\ Cons(A,\ Nil)))} \xrightarrow{\mathtt{Mem2}} \uparrow$$

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.

$t \xrightarrow{\ell : l \to r} t'$	if $\exists \sigma : \sigma(l) = t \wedge \sigma(r) = t'$
$t \xrightarrow{\ell : l \to r} \uparrow$	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 `Mem1` and `Mem3` is the following:

$$\begin{aligned}
&\mathtt{Member(A,\ Cons(B,\ Cons(C,\ Nil)))}\\
\xrightarrow{\mathtt{Mem3}}\ &\mathtt{Member(A,\ Cons(C,\ Nil))}\\
\xrightarrow{\mathtt{Mem3}}\ &\mathtt{Member(A,\ Nil)}\\
\xrightarrow{\mathtt{Mem1}}\ &\mathtt{False}
\end{aligned}$$

$$t \xrightarrow{\epsilon} t \qquad \frac{t \xrightarrow{s} t'}{t \xrightarrow{\mathsf{test}\, s} t} \qquad \frac{t \xrightarrow{s} \uparrow}{t \xrightarrow{\neg s} t}$$

$$\frac{t \xrightarrow{s_1} t' \quad t' \xrightarrow{s_2} t''}{t \xrightarrow{s_1 ; s_2} t''}$$

$$\frac{t \xrightarrow{s_1} t'}{t \xrightarrow{s_1 + s_2} t'} \qquad \frac{t \xrightarrow{s_2} t'}{t \xrightarrow{s_1 + s_2} t'}$$

$$\frac{t \xrightarrow{s_1} t'}{t \xrightarrow{s_1 <\!\!+ s_2} t'} \qquad \frac{t \xrightarrow{s_1} \uparrow \quad t \xrightarrow{s_2} t'}{t \xrightarrow{s_1 <\!\!+ s_2} t'}$$

$$\frac{t \xrightarrow{s[x := \mu x(s)]} t'}{t \xrightarrow{\mu x(s)} t'}$$

(a) positive rules

$$t \xrightarrow{\delta} \uparrow \qquad \frac{t \xrightarrow{s} \uparrow}{t \xrightarrow{\mathsf{test}\, s} \uparrow} \qquad \frac{t \xrightarrow{s} t'}{t \xrightarrow{\neg s} \uparrow}$$

$$\frac{t \xrightarrow{s_1} \uparrow}{t \xrightarrow{s_1 ; s_2} \uparrow} \qquad \frac{t \xrightarrow{s_1} t' \quad t' \xrightarrow{s_2} \uparrow}{t \xrightarrow{s_1 ; s_2} \uparrow}$$

$$\frac{t \xrightarrow{s_1} \uparrow \quad t \xrightarrow{s_2} \uparrow}{t \xrightarrow{s_1 + s_2} \uparrow}$$

$$\frac{t \xrightarrow{s_1} \uparrow \quad t \xrightarrow{s_2} \uparrow}{t \xrightarrow{s_1 <\!\!+ s_2} \uparrow}$$

$$\frac{t \xrightarrow{s[x := \mu x(s)]} \uparrow}{t \xrightarrow{\mu x(s)} \uparrow}$$

(b) negative rules

Table 2: Operational semantics for basic combinators.

A strategy is an algorithm for exploring the reduction graph induced by a set of rules. 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* $s_1 ; s_2$ of two strategies. It first applies s_1 and, if that succeeds, it applies s_2. For example, the reduction path above is described by the strategy `Mem3` ; `Mem3` ; `Mem1`.

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 `Mem1` + `Mem2` applies either `Mem1` or `Mem2`. Note that due to this operator there can be more than one way in which a strategy can succeed.

With the non-deterministic choice operator the programmer has no control over which strategy is chosen. The *deterministic* or *left choice* operator $s_1 <\!\!+ s_2$ is biased to choose its left argument first. It will consider the second strategy only if the first does not succeed. This operator can be used to give higher priority to rules. For example, rule `Mem2` and `Mem3` are overlapping rules. To express that `Mem3` should be applied only after it is certain that `Mem2` does not apply, the strategy

$$(\texttt{Mem1} + \texttt{Mem2}) <\!\!+ \texttt{Mem3}$$

can be used.

Strategies that repeatedly apply some rules can be defined using the *recursion* operator $\mu x(s)$. For instance, the strategy

$$\mu x((\texttt{Mem1} + \texttt{Mem2}) <\!\!+ (\texttt{Mem3} ; x))$$

repeatedly applies rule `Mem3` (if possible) until either rule `Mem1` or `Mem2` is applicable. The strategy fails if neither `Mem1` nor `Mem2` is ever applicable. (Note that ; has higher precedence than + and $<\!\!+$. Therefore, $(\texttt{Mem1} + \texttt{Mem2}) <\!\!+ (\texttt{Mem3} ; x)$ could also be written as $(\texttt{Mem1} + \texttt{Mem2}) <\!\!+ \texttt{Mem3} ; x$.)

The *identity* strategy ϵ always succeeds. It is often used in conjunction with left choice to build an optional strategy: $s <\!\!+ \epsilon$ 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 $\texttt{test}\, s$ 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 $\varphi(x_1, \ldots, x_n) = s$ introduces a new n-ary strategy operator φ. An application $\varphi(s_1, \ldots, s_n)$ of φ to n strategies denotes the instantiation $s[x_1 := s_1 \ldots x_n := s_n]$ of the body s 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

$$\texttt{repeat}(s) = \mu x((s ; x) <\!\!+ \epsilon)$$
$$\texttt{repeat1}(s) = s ; \texttt{repeat}(s)$$

The strategy $\texttt{repeat}(s)$ applies s as many times (zero or more) as possible. The strategy $\texttt{repeat1}(s)$ is like `repeat`, except that it must succeed at least once. Using `repeat`, we can define the strategy $\texttt{repeat}((\texttt{Mem1} + \texttt{Mem2}) <\!\!+ \texttt{Mem3})$ which is equivalent to $\mu x((((\texttt{Mem1} + \texttt{Mem2}) <\!\!+ \texttt{Mem3}) ; x) <\!\!+ \epsilon)$. It applies rules `Mem1`, `Mem2` and `Mem3` as often as possible and will always succeed.

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

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

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

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

$$\frac{\exists j \forall i : i,j \in \{1..n\} \wedge P(t_i) = \begin{cases} t_i' & \text{if } t_i \xrightarrow{s} t_i' \\ t_i & \text{if } t_i \xrightarrow{s} \uparrow \wedge\, i \neq j \end{cases}}{f(t_1,\dots,t_n) \xrightarrow{\boxtimes(s)} f(P(t_1),\dots,P(t_n))}$$

(a) positive rules

$$f(t_1,\dots,t_n) \xrightarrow{i(s)} \uparrow \quad \text{if } i > n$$

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

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

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

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

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

$$\frac{t_1 \xrightarrow{s} \uparrow \quad \dots \quad t_n \xrightarrow{s} \uparrow}{f(t_1,\dots,t_n) \xrightarrow{\boxtimes(s)} \uparrow}$$

(b) negative rules

Table 3: Operational semantics for term traversal operators. These rules are schemata that define a rule for each $f \in \Sigma$. In all rules $n \equiv \Sigma(f)$, $m \equiv \Sigma(g)$ and $1 \leq i \leq n$.

Backtracking Operationally, the non-deterministic choice operator $s_1 + s_2$ randomly chooses one strategy to apply and if that fails backtracks and attempts to apply the other one. However, backtracking is local only; if the first strategy succeeds the second will never be attempted. If both s_1 and s_2 succeed, then the order in which they are tried can affect the outcome of the larger strategy that encompasses the choice. For example, suppose that $t \xrightarrow{s_1} t'$ and $t \xrightarrow{s_2} t''$, but $t' \xrightarrow{s_3} \uparrow$ and $t'' \xrightarrow{s_3} t'''$. Then either $t \xrightarrow{(s_1+s_2);s_3} t'''$ or $t \xrightarrow{(s_1+s_2);s_3} \uparrow$ depending on the choice made for $s_1 + s_2$. The left choice operator $<\!+$ is also a local backtracking operator, but the order in which the alternatives are tried is fixed. Therefore, a strategy composed without $+$ is deterministic and either fails or succeeds.

2.4 Term Traversal

The operators introduced above apply strategies to the root of a term. This is not adequate for achieving all transformations. For instance, consider the extension of our list calculus with a concatenation operator `Conc`:

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

Application of rule `Cnc2` leads to an opportunity to apply these rules below the root of the term. For example, consider the reduction path:

$$\begin{array}{l} \texttt{Conc(Cons(1, Nil), Cons(2, Nil))} \\ \xrightarrow{\texttt{Cnc2}} \texttt{Cons(1, Conc(Nil, Cons(2, Nil)))} \\ \xrightarrow{2(\texttt{Cnc1})} \texttt{Cons(1, Cons(2, Nil))} \end{array}$$

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

In general, we want to be able to apply transformation rules at arbitrary depth in a term. For this purpose we introduce five basic operators for applying a transformation to the children of a constructor. The operational semantics of these operators is defined in Table 3. In conjunction with the operators defined above, these are sufficient to define a wide range of full term traversals.

The fundamental operation for term traversal is the application of a strategy to a specific child of a term. The strategy $i(s)$ applies strategy s to the i-th child. If it succeeds the child is replaced with the result of the transformation. If it fails the application of $i(s)$ fails. It also fails if i is greater than then the arity of the constructor of the term to which it is applied. We saw an example above, $2(\texttt{Cnc1})$ applies rule `Cnc1` to the second argument of the root.

Congruence operators specify application of strategies to the children of terms constructed with a specific constructor. For each term constructor $f \in \mathsf{dom}(\Sigma)$ there is a corresponding strategy operator f with arity $\Sigma(f)$. If $s_1, ..., s_{\Sigma(f)}$ are strategies then $f(s_1, ..., s_{\Sigma(f)})$ is the strategy that applies only to terms t with outermost constructor f and applies each s_i to the i-th child of t. For example, the strategy $\texttt{Cons}(s_1, s_2)$ applies to `Cons` terms and applies s_1 to the head of the list and s_2 to the tail. In the example above $\texttt{Cons}(\epsilon, \texttt{Cnc1})$ is equivalent to $2(\texttt{Cnc1})$. To apply the concatenation rules until the application of `Conc` is eliminated we can use the strategy

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

The strategy either terminates with rule `Cnc1` or else applies rule `Cnc2` and then recursively applies the strategy to the `Conc` created in the tail of the list. Another 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 traverse arbitrary data structures we introduce the operators $\Box(s)$, $\Diamond(s)$ and $\boxtimes$. These operators are defined generically on all terms over a signature Σ.

The strategy $\Box(s)$ applies s to each child of the root and succeeds if s succeeds for each child. It fails if s fails for one or more of the children. In case of constants, i.e., constructors without arguments, the strategy always succeeds, since there are no children. (As a consequence the strategy $\Box(\delta)$ succeeds exactly on constants.) 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):

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

For example, the strategy $\texttt{topdown}((\texttt{Cnc1}+\texttt{Cnc2}) \leftarrow\!\!\!+ \epsilon)$ tries to apply the rules Cnc1 and Cnc2 everywhere in a term in a topdown traversal. It always succeeds because of the escape $\leftarrow\!\!\!+\epsilon$.

The strategy $\Diamond(s)$ is the dual of $\Box(s)$; It applies s nondeterministically to one child for which it succeeds. It fails if there is no child for which it succeeds. In particular, it fails for constants, since they have no child for which s can succeed. As a consequence the strategy $\Diamond(\epsilon)$ succeeds exactly on non-constants. The duals of the pre- and post-order traversals defined above apply a strategy s *exactly once* in a term while traversing the term in a top-down or bottom-up order:

$$\begin{aligned} \texttt{oncetd}(s) &= \mu x(s \leftarrow\!\!\!+ \Diamond(x)) \\ \texttt{oncebu}(s) &= \mu x(\Diamond(x) \leftarrow\!\!\!+ s) \end{aligned}$$

The strategy oncetd(s) first tries to apply s at the root and terminates if that succeeds. Otherwise if s fails on the root, it tries to apply the strategy to one of the children. The strategy oncebu(s) first tries to find an application of s below the root. If that fails s must succeed at the root. For instance, the strategy $\texttt{oncetd}(\texttt{Cnc1}+\texttt{Cnc2})$ succeeds if it finds an application of either rule Cnc1 or Cnc2 in the term and fails otherwise.

Finally, $\boxtimes(s)$ is a hybrid of $\Box(s)$ and $\Diamond(s)$ that applies s to some children. 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 from $\Box$ is that it does not have to succeed for all children. The analogue of oncebu with $\boxtimes$ is the strategy somebu, defined as:

$$\texttt{somebu}(s) = \mu x(\boxtimes(x) \leftarrow\!\!\!+ s)$$

Where oncebu finds a single subterm for which s succeeds, somebu finds as many subterms as possible to which s applies, but at least one. However, as soon as s succeeds for a subterm t' of t, s is not applied to any of the nodes in the spine from t to t'. A version of this strategy that finds still more subterms to apply to is manybu, defined as:

$$\texttt{manybu}(s) = \mu x((\boxtimes(x)\,;(s \leftarrow\!\!\!+ \epsilon)) \leftarrow\!\!\!+ s)$$

After applying s to as many subterms as possible with $\boxtimes(x)$, s is also tried at the root. If s did not succeed on any subterm, it has to succeed on the root for the strategy to succeed. The analogous pre-order strategies are:

$$\begin{aligned} \texttt{sometd}(s) &= \mu x(s \leftarrow\!\!\!+ \boxtimes(x)) \\ \texttt{manytd}(s) &= \mu x((s\,;\Box(x \leftarrow\!\!\!+ \epsilon)) \leftarrow\!\!\!+ \boxtimes(x)) \end{aligned}$$

These strategies perform a single 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.

$$\texttt{innermost'}(s) = \mu x(\Box(x)\,;((s\,;x) \leftarrow\!\!\!+ \epsilon))$$

It first normalizes all subterms ($\Box(x)$), i.e., all strict 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. Using the other traversal strategies defined above a wide range of alternative normalization strategies can be defined. See also [27] for examples of alternative evaluation strategies.

3 Case Study: RML Optimizer

RML [26] is a strict functional language, essentially similar to the core of Standard ML [22] 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. The transformations are intended to improve the performance of programs either directly (e.g., (Dead1) and (Dead2), which perform dead code elimination) or by enabling future improving transformations (e.g., (Hoist1) and (Hoist2), which sequentialize code). 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, 13, 24]. These particular rules were inspired by those presented in [4]. In the sequel, we concentrate on the details of the implementation of these rules.

In these rules we use the following notation and auxiliary notions: We write $\vec{a}$ for a list of phrases $a_1 \ldots a_n$ with the appropriate separator for the list type. The function vars produces the set of free variables of an expression. An expression is *safe* if it contains no calls to side-effecting primitives or to user-defined functions; any safe function is guaranteed to be pure and terminating. An expression is *small* if

let $x : t =$ let $y : t' = e_0$ in e_1 in e_2	$\longrightarrow$ let $y : t' = e_0$ in let $x : t = e_1$ in e_2 if $y \notin$ vars(e_2)	(Hoist1)
let $x : t =$ letrec $\vec{fdec}$ in e_1 in e_2	$\longrightarrow$ letrec $\vec{fdec}$ in let $x : t = e_1$ in e_2 if for each $f : t(\vec{x}) = e'$ in $\vec{fdec}$: $f \notin$ vars(e_2)	(Hoist2)
let $x : t = e_1$ in e_2	$\longrightarrow e_2$ if $x \notin$ vars(e_2) and e_1 is safe	(Dead1)
letrec $\vec{fdec}$ in e	$\longrightarrow e$ if for each $f : t\ (\vec{x}) = e'$ in $\vec{fdec}$: $f \notin$ vars(e)	(Dead2)
let $x : t = se$ in e	$\longrightarrow$ let $x : t = se$ in $e\{se/x\}$	(Prop)
letrec $f : t\ (\vec{x}) = e'$ in $e[f(\vec{se})]$	$\longrightarrow$ letrec $f : t\ (\vec{x}) = e'$ in e[rename$(e'\{\vec{se}/\vec{x}\})$] if $f \notin$ (vars$(\vec{se}) \cup$ vars$(e[_])$) or e' is small	(Inline)
let $x : t = (se_1, \ldots, se_n)$ in e[select(i, x)]	$\longrightarrow$ let $x : t = (se_1, \ldots, se_n)$ in $e[se_i]$	(Select)
let $f : \vec{t} \to t = e_1$ in e_2	$\longrightarrow$ letrec $f : \vec{t} \to t\ (\vec{x}) =$ let $f' : \vec{t} \to t = e_1$ in $f'(\vec{x})$ in e_2 if $\lvert\vec{x}\rvert = \lvert\vec{t}\rvert$, f' and the x_i are fresh variables and e_1 is safe	(EtaExp)

Table 5: Transformation rules for RML

$$
\begin{array}{rll}
t & ::= b \mid t_1 \to t_2 \mid t_1, \cdots, t_n & \text{(Types)} \\
se & ::= x \mid c & \text{(Simple expressions)} \\
fdec & ::= f : t\ (x_1, \ldots, x_n) = e & \text{(Function declarations)} \\
vdec & ::= x : t = e & \text{(Variable bindings)} \\
e & ::= se & \text{(Expressions)} \\
 & \mid x(se_1, \ldots, se_n) & \\
 & \mid d(se_1, \ldots, se_n) & \\
 & \mid (se_1, \ldots, se_n) & \\
 & \mid \text{select}(i, se) & \\
 & \mid \text{let } vdec \text{ in } e & \\
 & \mid \text{letrec } fdec_1 \cdots fdec_n \text{ in } e &
\end{array}
$$

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. No variable is bound twice.

Table 4: Syntax of RML

it contains no nested declarations; inlining a function whose body is small cannot increase the size of the program (measured in number of expressions).

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. (The conditions of these rules are redundant in case no variable is bound twice.) All the other rules use features that are not provided by basic rewrite systems.

(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 occur in the body. (Prop), which performs constant and variable propagation, requires *substitution* of free occurrences of a variable by an expression. (Inline) is a context-sensitive rule which replaces 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* $e[f(\vec{se})]$. Inlining should only occur if f appears only once in e (expressed here as $f \notin$ (vars$(\vec{se}) \cup$ vars$(e[_])$)) or its body is small. (Inline) uses *simultaneous substitution* of a list of expressions for a list of variables. Furthermore, the rule *renames* all occurrences of bound variables with fresh variables, to preserve the invariant that no variable is bound twice. This invariant simplifies substitution and testing for the occurrence of a variable in an expression. Finally, (EtaExp) requires the generation of variables that are fresh with respect to the entire program.

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 variable 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 is 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 a strategy that traverses the subterm that matches c and applies rule $l' \to r'$. Also, checking that some term t_1 occurs as a subterm of a term t_2 requires traversing t_2. Therefore, instead of creating more complex primitives such 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 into strategy expressions.

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

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

$$\frac{t : \mathcal{E} \xrightarrow{s} t' : \mathcal{E}'}{t : \mathcal{E} \xrightarrow{\mathsf{where}\ s} t : \mathcal{E}'}$$

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

$$\frac{t_1 : \mathcal{E}_0 \xrightarrow{s_1} t_1' : \mathcal{E}_1 \quad \ldots \quad t_n : \mathcal{E}_{n-1} \xrightarrow{s_n} t_n' : \mathcal{E}_n}{f(t_1, \ldots, t_n) : \mathcal{E}_0 \xrightarrow{f(s_1, \ldots, s_n)} f(t_1', \ldots, t_n') : \mathcal{E}_n}$$

$$\frac{t_1 : \mathcal{E}_0 \xrightarrow{s} t_1' : \mathcal{E}_1 \quad \ldots \quad t_n : \mathcal{E}_{n-1} \xrightarrow{s} t_n' : \mathcal{E}_n}{f(t_1, \ldots, t_n) : \mathcal{E}_0 \xrightarrow{\Box(s)} f(t_1', \ldots, t_n') : \mathcal{E}_n}$$

$$\frac{\exists j \forall i : i, j \in \{1..n\} \wedge P(t_i) = \begin{cases} t_i' & \text{if } t_i : \mathcal{E}_{i-1} \xrightarrow{s} t_i' : \mathcal{E}_i \\ t_i & \text{if } t_i : \mathcal{E}_{i-1} \xrightarrow{s} \uparrow \wedge \\ & \quad \mathcal{E}_i = \mathcal{E}_{i-1} \wedge i \neq j \end{cases}}{f(t_1, \ldots, t_n) : \mathcal{E}_0 \xrightarrow{\boxtimes(s)} f(P(t_1), \ldots, P(t_n)) : \mathcal{E}_n}$$

(a) positive rules

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

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

$$\frac{t : \mathcal{E} \xrightarrow{s} \uparrow}{t : \mathcal{E} \xrightarrow{\mathsf{where}\ s} \uparrow}$$

$$\frac{t : (\mathcal{E} \backslash \vec{x}) \xrightarrow{s} \uparrow}{t : \mathcal{E} \xrightarrow{\{\vec{x}:s\}} \uparrow}$$

$$\frac{t_1 : \mathcal{E}_0 \xrightarrow{s_1} t_1' : \mathcal{E}_1 \quad \ldots \quad t_i : \mathcal{E}_{i-1} \xrightarrow{s_n} \uparrow}{f(t_1, \ldots, t_n) : \mathcal{E}_0 \xrightarrow{f(s_1, \ldots, s_n)} \uparrow}$$

$$\frac{t_1 : \mathcal{E}_0 \xrightarrow{s} t_1' : \mathcal{E}_1 \quad \ldots \quad t_i : \mathcal{E}_{i-1} \xrightarrow{s} \uparrow}{f(t_1, \ldots, t_n) : \mathcal{E}_0 \xrightarrow{\Box(s)} \uparrow}$$

$$\frac{t_1 : \mathcal{E} \xrightarrow{s} \uparrow \quad \ldots \quad t_n : \mathcal{E} \xrightarrow{s} \uparrow}{f(t_1, \ldots, t_n) : \mathcal{E} \xrightarrow{\boxtimes(s)} \uparrow}$$

(b) negative rules

Table 6: Operational semantics for environment operators.

Match, Build and Scope We first 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 producing a binding of subterms to the variables in l. Then 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 $\mathsf{match}(l) ; \mathsf{build}(r)$. 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 $\mathsf{dom}(\mathcal{E}') = \mathsf{dom}(\mathcal{E}) \cup \mathrm{vars}(t)$.

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 are defined in Table 6. The change in the format of the operational semantics should be reflected in the semantics of the operators introduced earlier. In the remainder of the paper the rules in Tables 2 and 3 should be read as follows: a transition $t \xrightarrow{s} t'$ denotes a transition $t : \mathcal{E} \xrightarrow{s} t' : \mathcal{E}'$. The only exceptions are the rules for congruence, $\Box$ and $\boxtimes$. See Table 6 for their definitions in the extended semantics.

Once a variable is bound it cannot be rebound to a different term. To use a variable name more than once we introduce variable scopes. A scope $\{\vec{x} : s\}$ locally undefines the variables $\vec{x}$. That is, the binding to a variable x_i outside the scope $\{\vec{x} : s\}$ is not visible inside it, nor is the binding to x_i inside the scope visible outside it. The notation $\mathcal{E} \setminus \vec{x}$ denotes $\mathcal{E}$ without bindings for variables in $\vec{x}$. $\mathcal{E}|\vec{x}$ denotes $\mathcal{E}$ restricted to $\vec{x}$. The strategy operator where is similar to the test operator of Section 2 in that it tries a strategy and returns the original term if it succeeds. However, it keeps the transformation on the environment. This operator can be used to encode a local computation that binds the answer to a variable to be used outside it, without actually transforming the term.

Note that this definition supports matching with non-linear patterns. If a variable x occurs more than once in a pattern t, then $\mathsf{match}(t)$ succeeds only if all occurrences of x in t are bound to the same term. Moreover, if a variable x in t was already bound by a previous match, it should match to the exact same term as it was bound to before. For example, consider the strategy `in` that tests whether x is a subterm of y. It is defined as

$$\begin{aligned} \mathtt{in} = \{x, y : \ & \mathsf{match}((x, y)) \, ; \\ & \mathsf{test}(\mathsf{build}(y) \, ; \mathsf{oncetd}(\mathsf{match}(x)))\} \end{aligned}$$

The first match matches a pair of terms (t_1, t_2), binding t_1 to x and t_2 to y. The build replaces the pair by t_2. The traversal `oncetd` searches for an occurrence of t_1 in t_2 by matching x (which is bound to t_1) against subterms of t_2. The strategy succeeds if it actually finds a matching subterm. The use of `test` ensures that the predicate does not affect the term to which it is applied.

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 $\ell : l \rightarrow r$ translates to a strategy definition

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

It introduces a local scope for the variables $\text{vars}(l, r)$ used in the rule, 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 of rewrite rules to achieve subcomputations. A rule

$$\ell : l \rightarrow r \;\mathsf{where}\; s$$

corresponds to the strategy

$$\ell = \{\text{vars}(l, r, s) : \mathsf{match}(l) \, ; \mathsf{where}(s) \, ; \mathsf{build}(r)\}$$

that first matches l, then tests s and finally builds r. The strategy s can be any strategy that affects the environment in order to bind variables used in r or just tests a property of the left-hand side. Note that s can transform the original term, but the effect of this is canceled by the where. Only the side-effect of s on the environment matters.

Boolean Conditions Conditions that check whether some predicate holds are implemented 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. 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 choice. In such conditions we use the notation $\langle s \rangle \; t$, which corresponds to $\mathsf{build}(t) \, ; s$. For instance, the encoding of the dead code elimination rule (Dead1) is:

$$\begin{array}{l} \texttt{Dead1} : \texttt{Let}(\texttt{Vdec}(x, t, e_1), e_2) \rightarrow e_1 \\ \qquad \mathsf{where}\; \neg(\langle \texttt{in} \rangle \; (\texttt{Var}(x), e_2)) \, ; \langle \texttt{safe} \rangle \; e_1 \end{array}$$

Where $\langle \texttt{in} \rangle \; (t_1, t_2)$ tests that t_1 is a subterm of t_2 as defined before and safe tests that an expression is terminating and side-effect free.

Matching Condition Another kind of subcomputation is the application of a strategy to a term built with variables from the left-hand side l, matching the result against a pattern with variables used in the right-hand side r. The notation $s \Rightarrow t'$ is a shorthand for $s \, ; \mathsf{match}(t')$. The combined notation $\langle s \rangle \; t \Rightarrow t'$ thus stands for $\mathsf{build}(t);s;\mathsf{match}(t')$. It first applies s to t and then matches the result against t' binding its variables.

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

$$\ell : l \rightarrow r \, [x] \;\mathsf{where}\; \langle s \rangle \; t \Rightarrow x$$

where x is a new variable. The notation $t \, [t']$ denotes a meta-context, i.e. a term t with a specific occurrence of a subterm t'. The replacement by t'' of the subterm t' in t is denoted by $t \, [t'']$.

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

$$\ell : l \; [\![c[l']]\!] \rightarrow r \; [\![c[r'](\varphi)]\!]$$

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

$$\begin{array}{l} \ell : l \, [c] \rightarrow r \, [c'] \\ \quad \mathsf{where}\; \langle \varphi(\{\text{vars}(l', r') \backslash \text{vars}(l \, [c], r \, [c']) : \\ \qquad\qquad\qquad \mathsf{match}(l') \, ; \mathsf{build}(r')\}) \rangle \; c \Rightarrow c' \end{array}$$

where c' is a fresh variable. The strategy in the where clause traverses the subterm matching c to find one or more occurrences of l' and replaces them with r'. The result of the traversal is bound to c', which is then used in the right-hand side of the rule. Note that the variables of l' and r' are locally scoped except those common with the variables of l and r, since those are instantiated in l and/or used in r. The strategy operator φ that is specified in conjunction with the contexts indicates the strategy used for the traversal. This determines where and how often the rule is applied.

As an example, consider the encoding of the (Select) rule:

$$\begin{array}{l} \texttt{Sel} : \texttt{Let}(\texttt{Vdec}(x, t, ses), e[\texttt{Select}(i, x)]) \rightarrow \\ \qquad \texttt{Let}(\texttt{Vdec}(x, t, ses), e[\langle \texttt{index} \rangle \; (i, ses)](\texttt{sometd})) \end{array}$$

It uses the traversal strategy sometd to replace all occurrences of $\texttt{Select}(i, x)$ in e by the corresponding element of the record. The strategy index takes the i-th element from the list *ses* of simple expresions. Using the encoding defined above this rule translates to the rule:

$$\begin{array}{l} \texttt{Sel} : \texttt{Let}(\texttt{Vdec}(x, t, ses), e) \rightarrow \texttt{Let}(\texttt{Vdec}(x, t, ses), e') \\ \quad \mathsf{where}\; \langle \texttt{sometd}(\{i : \mathsf{match}(\texttt{Select}(i, x)) \, ; \\ \qquad\qquad\qquad\qquad \mathsf{build}(\langle \texttt{index} \rangle \; (i, ses))\}) \rangle \; e \Rightarrow e' \end{array}$$

Note that the variable i is local to the context traversal and can thus be instantiated to more than one value.

We have only discussed rules with one context. Rules with more than one context are beyond the scope of this paper.

5.4 Variable 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, the following rules map the variable binding constructs of RML to the list of variables they bind.

$$
\begin{aligned}
&\texttt{Bind1} : \texttt{Let}(\texttt{Vdec}(\_, x, \_), \_) \rightarrow [x];\\
&\texttt{Bind2} : \texttt{Letrec}(\mathit{fdecs}, \_) \rightarrow\\
&\qquad \langle \mathsf{map}(\{f : \mathsf{match}(\texttt{Fdec}(\_, f, \_, \_)); \mathsf{build}(f)\}) \rangle\ \mathit{fdecs}\\
&\texttt{Bind3} : \texttt{Fdec}(\_, \_, xs, \_) \rightarrow xs
\end{aligned}
$$

Given these rules and a couple of additional rules for indicating in which arguments the constructs are binding (see Appendix A) the strategy `rename` renames all bound object variables. This strategy uses the built-in strategy `new` which generates fresh names. See Appendix D for its definition.

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 defines the abstract syntax of the object language of the optimizer. The rules section defines the individual transformation rules. The strategies section defines two strategies for combining these rules into an optimization algorithm. The module imports several auxiliary modules that are defined in the appendices.

Observe that the specification of the rules is very close to the original rules in Table 5. The main difference is that the inline rule has been split into two rules. Rule `Inl1` handles the case that the body of the function is small and hence can be inlined everywhere in the body of the `Letrec`. Rule `Inl2` has no condition and replaces exactly one occurence of an application of the function `f` in the body of the `Letrec`. To achieve the condition that this rule should only be applied when `f` does not occur in the body of the `Letrec` after inlining the rule is always followed by an application of `Dead2`. If `Dead2` succeeds this guarantees that `f` does not occur anymore.

Strategies An advantage of our approach of separating the specification of rules from strategies is the ability to experiment with alternative strategies for the same set of rules. We present the strategies `optimize1` and `optimize2` for the RML transformation rules.

In `optimize1` and `optimize2`, we have avoided applying `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 `manydownup` (see Appendix B) to apply the rules.

The strategy `manydownup` applies a strategy `s` at all positions of a term once while going down into the term and once on the way back. It fails when none of these applications succeed. If it succeeds we know that some redex has been reduced. Hence, we can repeat `manydownup` to normalize a term.

While `optimize1` uses generic strategies, `optimize2` uses the properties of the rules in order to apply them in a more restricted way. 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 try to reapply the strategy to this term.

7 Implementation

The strategy language presented in this paper has been implemented in Standard ML. 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 programming environment contains a compiler that generates C code. 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 backtracking. The run-time environment of compiled strategies is based on the ATerm C-library [23]. 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. No performance results are available yet.

The compiler implements a straightforward translation of strategy expressions to C programs that performs no optimizations. Currently we are bootstrapping the compiler by specifying it in the strategy language itself. This gives us the opportunity to apply optimizations to strategies. There are several levels of optimization we are considering: simplification of expressions by applying simple algebraic laws; factoring out common prefixes from the alternatives of choices; propagating knowledge about matching history through traversals. Finally, it is worth considering the automatic derivation of more refined strategies by specializing applications of generic strategies to specific rules. An example would be to derive a strategy in the style of `optimize2` from a strategy in the style of `optimize1` in Table 7.

An alternative approach to implementation of the language would be as a library of functions in a general purpose language, e.g., a functional language such as ML or Haskell. For each operator in the core language a corresponding function is defined. In fact, our interpreter uses such a library. The advantage of such an embedded implementation is that work on run-time environment and such can be borrowed from the host language. However, since a more general framework is used, the host compiler cannot take advantage of knowledge of the specific domain of term transformation.

8 Related Work

Program Optimization There have been many attempts to build frameworks for program analysis and optimization, often using special-purpose formalisms. The systems closest to ours in spirit are probably OPTRAN [19] and Dora/Tess [12]. Like our system, these are based on ideas from term rewriting and emphasize separating the declarative specification of rewrite rules from the strategy to be used in applying them. Unlike our system, however, they support only a fixed set of pre-defined strategy options, and the same strategy must be used for all rules and for the whole tree. The options provided by the two systems are similar: traverse the tree top-down or bottom-up; traverse children left-to-right or right-to-left; rewrite each node only once per traversal or iterate at each node until a fixed point is reached. Our strategy language can easily implement these options (e.g., in a general-purpose library), but can also define much more fine-grained strategies where needed.

Numerous other systems provide mechanisms for generating transformation code, but none appears to offer our flexible combination of generic and rule-specific strategies. DFA&OPT-MetaFrame [18], Sharlit [25], Genesis [28], and OPTIMIX [5] are all primarily designed as *analyzer* generator systems, each focused on a particular style of analysis. Their published descriptions do not give many details about their transformation capabilities, but none appears to give the user any control over transformation order. At the opposite extreme, KHEPERA [11], TXL [9, 21], and Puma [15], provide succinct primitives for matching and building subtrees, but for the most part require that tree traversal be programmed explicitly in imperative style, node by node. TXL includes a "searching" version of the match operator which behaves like an application of our `oncetd` strategy. KHEPERA provides a built-in construct to iterate over the immediate children of a node.

Other recent approaches to high-level description of optimizations include Aspect-Oriented Programming [16], which advocates the use of domain-specific "aspect" languages to describe optimization of program IR trees (in practice, LISP is generally used), and Intentional Programming [2], which provides a library of routines for manipulating ASTs. Neither of these approaches particularly encourages separation of rules from strategies for their application.

Strategies In conventional term rewriting languages, rewrite systems are assumed to be confluent and terminating and therefore, strategies are only considered at the meta-level of language design and implementation. In particular, algebraic specification formalisms such as ASF+SDF [10] only provide one 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.

In theorem proving such fixed strategies are not sufficient since a theorem can be proved in many ways. The theorem proving framework LCF [14] introduced tactics for proving theorems. A tactic transforms a goal to be proved into a list of subgoals and a proof. By repeatedly applying tactics a list of goals is reduced to the empty list, which indicates that the original goal is proven. A series of basic tactics are provided, including a simplification tactic that applies a set of rewrite rules using a fixed strategy. New tactics can be formed from existing ones using tacticals. The standard tacticals are similar to our identity, sequential composition, left choice and repeat strategy operators, although they have a somewhat different semantics since they apply to a subgoal instead of to the root of a term. In the theorem proving domain there is no need for traversal tacticals.

In the specification formalism ELAN [17] the notion of transformation of goals to a list of subgoals is generalized to arbitrary term rewrite rules. Strategies are regular expressions over the set of rule labels. In [17] this approach is used to define constraint solvers that consist of rules that rewrite a list of constraint into a new list of constraints. A strategy repeatedly applies such rules until a solution is found. In later versions of the language, e.g., [7], the set of strategy operators is extended with congruence operators to support term traversal.

ELAN does not provide generic traversal operators analogous to our $i(s)$, $\Box$, $\Diamond$ and $\boxtimes$. Instead traversals have to be defined explicitly for each datatype using congruences. Recursive strategies are expressed in ELAN using recursive strategy definitions. Further differences with ELAN are the negation and test operators and the breakdown of rules into primitives. Where ELAN has a fixed syntax for rewrite rules our strategy language can easily be extended with expressive features that are implemented in terms of the core language.

Maude [8] 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.

The language described in this paper was inspired by the strategy language of ELAN. The first version was described in [20], which presents a strategy language with identity, sequential composition, choice, recursion, and a generic 'push-down' operator that is used to define $\Box$ and $\boxtimes$. Its design was guided by the process algebra ACP [6]. An interpreter for strategy expressions is specified in the algebraic specification formalism ASF+SDF [10]. Basic strategies are unconditional ASF+SDF rewrite rules.

In this paper we have extended our first language with the operators failure, negation, test, path, congruence and $\Diamond$. Furthermore, we cater for a much more expressive set of rules by means of the breakdown of rewrite rules into match, build and scope. In addition, our current language is implemented by compilation to C.

Technical contributions of our work in the setting of strategy languages include the modal operators $\Box$, $\Diamond$ and $\boxtimes$ that enable very concise specification of term traversal; a set general purpose traversal strategies; the explicit recursion operator $\mu x(s)$; the refinement of rewrite rules into match and build; and the encoding of complex rewrite rules into strategies, in particular the expression of rules with contexts.

In [27] we describe how our core strategy language can be used to implement conventional term rewriting engines and how these can be extended with non-standard evaluation strategies.

```
module rml
imports traversal
imports list
imports substitution
imports renaming
imports rml-aux
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, x, Let(vdec, e1)), e2) -> Let(vdec, Let(Vdec(t, x, e1), e2))

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

Dead1  : Let(Vdec(t, x, e1), e2) -> e2 where not(<in> (Var(x), e2)); <safe> e1

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

Prop   : Let(Vdec(t, x, Simple(se)), e[Var(x)]) -> Let(Vdec(t, x, Simple(se)), e[se](sometd))

Inl1   : Letrec([Fdec(t, f, xs, e1)], e2[App(Var(f), ss)]) ->
         Letrec([Fdec(t, f, xs, e1)], e2[<rsubs; rrename> (xs, ss, e1)](sometd))
         where <small> e1

Inl2   : Letrec([Fdec(t, f, xs, e1)], e2[App(Var(f), ss)]) ->
         Letrec([Fdec(t, f, xs, e1)], e2[<rsubs; rrename> (xs, ss, e1)](oncetd))

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

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), ses)))], e2)
         where <safe> e1; new => f2; <map(new)> ts => xs; <map(MkVar)> xs => ses
strategies

opt1 = innermost'(Hoist1 + Hoist2);
       manydownup(((Inl1 <+ (Inl2; Dead2) + Sel + Prop); 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; x)
                 + Letrec(id, x); (Dead2 <+ try(Letrec(map(Fdec(id,id,id,x)),id);
                                                try((Inl1; try(Dead2) <+ Inl2; Dead2); x)))))
optimize2 = bottomup(try(EtaExp)); opt2
```

Table 7: Specification of RML transformation rules

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, or add a new transformation rule and inserting its tag at the appropriate place in the strategy; 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 [20] 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. We thank Eugenio Moggi and Philip Wadler for remarks on the semantics and Patricia Johann for remarks on a previous version of this paper.

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 *Proceedings 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 *Proceedings 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 Hélène Kirchner. Controlling rewriting by rewriting. In José Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar, Pacific Grove, CA, September 1996. Elsevier.

[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 and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89, Asilomar, Pacific Grove, CA, September 1996. Elsevier.

[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 *Proceedings USENIX Conference on Domain-Specific Languages*, pages 243–255, October 1997.

[12] Charles Donald Farnum. *Pattern-Based Languages for Prototyping of Compiler Optimizers.* PhD thesis, University of California, Berkeley, 1990. Technical Report CSD-90-608.

[13] 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.

[14] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF. A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1979.

[15] 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.

[16] 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.

[17] C. Kirchner, H. Kirchner, and M. Vittek. Implementing computational systems with constraints. In P. Kanellakis, J-L. Lassez, and V. Saraswat, editors, *Proceedings of the first Workshop on Principles and Practice of Constraint Programming*, pages 166–175, Providence R.I., USA, 1993. Brown University.

[18] Marion Klein, Jens Knoop, Dirk Koschützki, and Bernhard Steffen. DFA & OPT–METAFrame: A toolkit for program analysis and optimization. In *Proceedings of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 418–121, Passau (Germany), March 1996. Springer Verlag.

[19] Peter Lipps, Ulrich Möncke, and Reinhard Wilhelm. OPTRAN - a language/system for the specification of program transformations: System overview and experiences. In *Proceedings 2nd Workshop on Compiler Compilers and High Speed Compilation*, volume 371 of *Lecture Notes in Computer Science*, pages 52–65, October 1988.

[20] 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.

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

[22] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT press, 2nd edition, 1997.

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

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

[25] Steven W. K. Tjiang and John L. Hennessy. Sharlit—A tool for building optimizers. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, July 1992.

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

[27] Eelco Visser and Zine-el-Abidine Benaissa. A core language for rewriting. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications (WRLA'98)*, Electronic Notes in Theoretical Computer Science, Pont-à-Mousson, France, September 1–4 1998. Elsevier.

[28] Deborah Whitfield and Mary Lou Soffa. The design and implementation of Genesis. *Software– Practice and Experience*, 24(3):307–325, March 1994.

A Auxiliary Strategies for RML

The module `rml-aux` defines the predicates `small` and `safe`. Furthermore, it defines the strategies `rsubs` for substitution in RML expressions and `rrename` for renaming bound variables in RML expresions. These strategies are instantiations of the language independent strategies `subs` (Appendix D) and `rename` (Appendix E).

```
module rml-aux
imports substitution
imports renaming
strategies

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

  safe = not(oncetd(App(id,id) +
                    match(Papp("assign", _))))

rules

  IsVar : Var(x) -> x

  MkVar : x -> Var(x)

  Bind1 : Let(Vdec(_,x,_),_) -> [x]
  Bind2 : Letrec(fdecs, _) ->
          <map({f : match(Fdec(_,f,_,_));
                    build(f)})> fdecs
  Bind3 : Fdec(_,_,xs,_) -> xs

  PutVar : (f, Fdec(t, f', xs, e)) ->
           Fdec(t, f, xs, e)

  PutVars(nvs, nbnd) :
    fdecs -> <zip(PutVar; nbnd)> (fs, fdecs)
    where nvs => fs

strategies

  rsubs = subs(IsVar)

  rn_apply(nvs, bnd, nbnd) =
      Let(Vdec(id, nvs; Hd, nbnd), bnd)
    + Fdec(id, id, nvs, bnd)
    + Letrec(PutVars(nvs, nbnd), bnd)

  rrename =
    rename(IsVar, MkVar, Bind1 + Bind2 + Bind3)
```

B Traversal Strategies

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

```
module traversal
strategies
  try(s)        = s <+ id

  repeat(s)     = rec x(try(s; x))
  repeat1(s)    = s; repeat(s)

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

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

  somebu(s)     = rec x(some(x) <+ s)
  sometd(s)     = rec x(s <+ some(x))

  manybu(s)     = rec x((some(x); try(s)) <+ s)
  manytd(s)     = rec x((s; all(try(x))) <+ some(x))
  manydownup(s) = rec x((s; all(try(x)); try(s))
                        <+ (some(x); try(s)))

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

  reduce(s)     = repeat(rec x(some(x) + s))
  outermost(s)  = repeat(oncetd(s))
  innermost(s)  = repeat(oncebu(s))
  innermost'(s) = rec x(all(x); try(s; x))

  in            = {x,y: match((x,y));
                        test(<oncetd(match(x))> y)}
```

C List Strategies

```
module list
signature
  operations
    Nil  : List(a)
    Cons : a * List(a) -> List(a)
rules

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

strategies

  index      = repeat(Ind2); Ind1
  map(s)     = rec x(Nil + Cons(s, x))
  at_tail(s) = rec x(Nil; s + Cons(id, x))
  fetch(s)   = rec x(Cons(s, id) <+ Cons(id, x))

rules

  conc : (l1, l2) -> <at_tail(build(l2))> l1

  lookup(mklst) : x -> y
                where mklst; fetch(match((x, y)))

  Zip0       : (Nil, Nil) -> Nil
  Zip1(a, b) : (Cons(x, xs), Cons(y, ys)) ->
               Cons(<a> (x, y), <b> (xs, ys))
strategies

  zip(s) = rec x(Zip0 + Zip1(s, x))
```

D Substitution

The strategy `subs(isvar)`, applied to a triple `(xs, ts, t)` of a list of strings `xs`, a list of terms `ts` and a term `t`, replaces each occurence in `t` of a variable `x` from `xs` by the corresponding term in `ts`; The parameter strategy `isvar` should be a rule (or choice of rules) that maps a term representing an (object) variable to its name. Typically such a rule is of the form `Var(x) -> x`.

The strategy first matches its arguments. Then it zips together the string list and the term list to get a table `tbl` that associates variable names with terms they have to be substituted with. (This fails if `xs` and `ts` are lists of different length, because `zip` will fail.) Finally, a traversal of the term `t` replaces each variable occuring in the table by its target. Note that the strategy `alltd` stops after it has found an application of its argument strategy. This is necessary to avoid applying the substitution to the terms being substituted. This strategy assumes that bound variables are renamed such that no variable is bound twice.

```
module substitution
imports traversal
imports list
rules

GetVar(mktbl)
  : x -> z where mktbl; fetch(match((x, z)))

strategies

subs(isvar) =
{tbl, xs, ts, t
 : match((xs, ts, t)) ;
   <zip(id)> (xs, ts) => tbl;
   <alltd(isvar; GetVar(build(tbl)))> t
}
```

E Renaming

The strategy `rename(isvar, mkvar, bnd)` renames all bound variables in a term to new variables. It is parameterized with strategies that express what the variables and the binding constructs of the language are. The parameter `isvar` recognizes variables and maps them to their name. The parameter `mkvar` maps a string to a variable. The parameter `bnd` maps each binding construct to the list of variables that it binds.

In addition, the user should specify the strategy operator `rn_apply(a, b, c)` such that for each variable binding construct `a` is applied to the subterm containing the variable(s) `b` is applied to the subterms in which the variables are bound `c` is applied to the subterms in which the variables are not bound. For an example, see Appendix A.

```
module renaming
imports traversal
imports list
strategies

rename(isvar, mkvar, bnd) =
{t : match(t); build((t, []))};
rec ren1
({l:
  {t: match((t, l)); build(t)};
  rec ren2
  (isvar; lookup(build(l)); mkvar
  <+ {xs, ys, l':
      where(bnd => xs; map(new) => ys;
            <conc> (<zip(id)> (xs, ys), l) => l');
      rn_apply(build(ys),
            {x: match(x); build((x, l'))}; ren1,
            {x: match(x); build((x, l))};  ren1)}
  <+ all(ren2))
})
```