First-class Rules and Generic Traversal

Eelco Dolstra
Eelco Visser

Institute of Information and Computing Science, Utrecht
University, Box 80089, 3508 TB Utrecht, The Netherlands
{eelco, visser}@cs.uu.nl

November 29, 2001

Abstract

In this paper we present a functional language supporting first-class rules and generic traversal. This is achieved by generalizing the pattern matching constructs of standard functional languages. The case construct that ties rules together and prevents their reuse, is replaced by separate, first-class, pattern matching rules and a choice combinator that deals with pattern match failure. Generic traversal is achieved through application pattern matching in which a constructor application is generically divided into a prefix and a suffix, thus giving generic access to the subterms of a constructor term. Many highly generic term traversals can be defined in a type-safe way using this feature.

These features support a direct and natural encoding of program transformation rules and strategies. The generalization of pattern matching subsumes several proposals for extensions of pattern matching such as views, guarded patterns, and transformational patterns.

1 Introduction

Program transformation techniques are used in a wide variety of applications including optimizing compilers, program normalizers, aspect weavers, obfuscation engines, refactoring browsers, and software renovation tools. The basic architecture of a transformation system is the same across all these applications. Parsers translate program text to an internal representation (abstract syntax) and unparsers translate abstract syntax back to text. In between, a transformation component modifies the abstract syntax tree to achieve the transformation. As with all programming, it is desirable to implement transformations at a high level of abstraction, reusing standard components or generating components from specifications. Where parsers and pretty-printers are generated from specifications, transformation components are usually implemented in general purpose languages without special support for transformation.

Term Rewriting Term rewrite systems [2] offer a good basis for the declarative specification and implementation of transformation systems. Algebraic terms are adequate representations for abstract syntax trees. Term rewrite rules directly encode single transformation steps on abstract syntax trees using term pattern matching to concisely characterize the subexpressions to transform. Term rewriting consists of exhaustively applying rewrite rules to a term until it is in normal form, i.e., does not contain any subterms to which a rule can be applied. Term rewriting systems such as ASF+SDF [12], ELAN [4], and Maude [10] implement this operation efficiently, usually employing an innermost strategy.

The advantage of rewrite systems is that transformation rules can be defined independently and that their application is automatic, i.e., no traversal over syntax trees has to be defined. However, sets of transformation rules for a programming language are usually non-confluent, i.e., different normal forms can be reached, and/or non-terminating, i.e., infinite reduction chains exist. This is usually solved by encoding control in auxiliary operators that control the application of transformation rules. This results in a style of term rewriting that can be characterized as first-order functional programming in which the advantages of rewriting are lost, i.e., rules are no longer independent, but tied to a particular transformation function, and the definition of transformation functions involves a considerable overhead due to the explicit definition of tree traversals.

Functional Programming Functional languages such as Haskell [21] and ML [20] offer some of the same ingredients as term rewriting systems, i.e., algebraic data types and pattern matching. In addition, higher-order functions and polymorphism allow the definition of traversal schemas such as folds for specific data types. However, transformation rules cannot be defined as first-class entities, but have to be combined using the case construct, thus limiting their reuse. The existence of several proposals for extensions of pattern matching such as views [28, 8], guarded patterns and transformational patterns [14], is an indication of the limitations imposed by the case construct. Furthermore, traversals cannot be defined generically across data types, since polymorphism restricts reuse to one shape. Polytypic extensions of functional languages [15, 16], allow the generic definition of functionality across data types, but do not support the definition of traversals that cross type boundaries *and* can be instantiated with different functions.

Rewriting Strategies Several extensions to rewriting have been proposed that try to combine the independent rules of rewriting with the need for control over the application of rewrite rules. In TAMPR [5] a transformation is divided into a sequence of normal form computations with respect to different sets of rewrite rules. In ASF+SDF traversal functions [7] reduce the overhead of the definition of traversals over syntax trees.

Rewriting strategies generalize such approaches by making the control over the

application of rewrite rules programmable by means of a language of strategy combinators. Rewriting strategies were introduced in the language ELAN [4]. Strategies were extended with primitives for term traversal in the program transformation language Stratego [27]. A few primitives allow the definition of a wide range of generic traversals. Rules are first-class and can be reused in different transformations and in different combinations. This paradigm supports concise formulation of transformation systems. Stratego has a few shortcomings, however. Even though there is a typeful way of defining and using strategies, the language is untyped, since typing generic traversals is beyond existing type systems. Although recent work addresses the typing of strategies in a many-sorted monomorphic setting [17], no type system for a polymorphic setting exists. Furthermore, the built-in traversal primitives are ad-hoc; a more fundamental way of defining traversal primitives is desirable. Finally, strategies in Stratego are not fully higher-order, but only second-order.

RhoStratego In this paper we present RhoStratego, a functional language supporting first-class rules and generic traversal. The language admits a direct and natural encoding of transformation rules and strategies. It combines the advantages of strategic programming (separation of rules and strategies) with the advantages of lazy higher-order functional programming (abstraction, definition of control constructs).

This is achieved by generalizing the pattern matching constructs of standard functional languages such as Haskell and ML. The case construct that ties rules together and prevents their reuse, is replaced by separate (first-class) pattern matching rules and a choice combinator that deals with pattern match failure. Generic traversal is achieved through application pattern matching in which a constructor application is generically divided into a prefix and a suffix, thus giving generic access to the subterms of a constructor term. Many highly generic term traversals can be defined using this feature. The generalization of pattern matching presented in this paper subsumes several proposals for extensions of pattern matching such as views [28, 8], guarded patterns, and transformational patterns [14].

The language is fully typed and has a type system in which type preserving and type unifying traversals can be typed. A prototype compiler for the language has been built and is available for experimentation.

Outline We start with an informal account of RhoStratego introducing first-class rules in Section 2 and generic traversal in Section 3. In Section 4 we present a rewriting semantics and a lazy evaluation strategy. The type system is described in Section 5. A brief overview of the implementation is given in Section 6. In Section 7 we discuss related work; in particular the encoding of extensions to pattern matching. We discuss future work in Section 8 and conclude in Section 9.

program	→	*decl**	(top level)
term	→	*var*	(variable)
	|	*constr*	(constructor)
	|	**fail**	(failure)
	|	*term term*	(application)
	|	`^`*term*	(cut)
	|	*pat* `->` *term*	(rule)
	|	*term* `<+` *term*	(choice)
	|	**let** *decl** **in** *term*	(let-bindings)
	|	(*term*)	(parentheses)
pat	→	*var*	(variable)
	|	*constr*	(constructor)
	|	*pat pat*	(application)
decl	→	*var* = *term*;	(binding)

Figure 1: RhoStratego abstract syntax

2 First-Class Rules

RhoStratego is a non-strict purely functional programming language. It consists essentially of the λ-calculus extended with constructors, pattern matching, let-bindings, and pattern match failure handling. In this section we introduce the syntax of the language and show how it supports first-class rules. In the next section we show how the language admits generic traversal.

The syntax of the language, shown in figure 1 in BNF notation, is similar to that of Haskell. Boldface and italics denote terminals and non-terminals, respectively. The productions are listed in order of decreasing priority. For example, `A -> B <+ C -> D` means `(A -> B) <+ (C -> D)`.

2.1 Lambda Rules

A rule x `->` t abstracts a variable x from a term t. Inspired by the notation used in term rewriting, there is no symbol that syntactically starts the abstraction (such as a λ or Haskell's backslash). An application `(`x `->` t_1`)` t_2 of a rule x `->` t_1 to a term t_2 amounts to the body t_1 in which the variable x is bound to the argument t_2. Thus, familiar functions such as the identity and constant functions are defined as follows:

```
id = x -> x;
const = x -> y -> x;
```

Naturally, arguments of rules can be other rules, allowing the definition of higher-order functions such as the composition operator:

```
. = f -> g -> x -> f (g x);
```

```
Var  :: String -> Exp
Num  :: Int -> Exp
Prim :: String -> Exp
Abs  :: String -> Exp -> Exp
App  :: Exp -> Exp -> Exp
Let  :: String -> Exp -> Exp -> Exp
```

Figure 2: Constructor declarations for the abstract syntax of a simple functional language.

2.2 Pattern Matching Rules

The abstract syntax of programs and other symbolic data can be represented by means of algebraic data types. For example, Figure 2 defines constructors for the abstract syntax of a simple functional language. Following Haskell's tradition, constructor names start with capitals. The details of the type system will be discussed in Section 5.

Values built with constructors can be deconstructed using pattern matching rules. An application `((`p `->` t_1`)` t_2`)` of a pattern matching rule `(`p `->` t_1`)` matches the argument t_2 against the pattern p, binding the variables in the pattern to the corresponding terms in t_2. As examples of pattern matching rules consider the simplification rules for expressions in Figure 3. Using these definitions, the term `plusZero (App (App (Prim "+") (Num 0)) (Var "z"))` reduces to `(Var "z")`.

2.3 Choosing between Values

An application `((`p `->` t_1`)` t_2`)` of a matching rule fails if the pattern p does not match the argument t_2. For example, the application `plusZero (Var "z")` fails. In general, an expression fails when a pattern match failure occurs somewhere in the evaluation of the expression.

In functional languages pattern match failure is handled by means of a case construct in which all alternative rules are combined. Alternatives are tried one by one until a successful match is found. It is generally a fatal error if there is no alternative that matches the subject value. The case construct ties rules together and makes it impossible to reuse rules in different contexts or to pass them on to a function.

RhoStratego inherits Stratego's left choice operator (`<+`) to handle pattern match failure. The term `(`t_1 `<+` t_2`)` evaluates to the value of t_1 unless its evaluation fails, in which case t_2 is chosen. For instance, if t stands for `(App (App (Prim "+") (Num 17)) (Num 42))`, the term `(plusZero` t `<+ plusFold` t`)` will first consider the application `(plusZero` t`)`. Since that fails the application `(plusFold` t`)` is evaluated, which results in `(Num 59)`.

Using the choice operator, individual rules can be combined in various ways to

form composite transformations. For example, the function `foldOne` performs one simplification step:

```
foldOne = x -> (plusZero x <+ plusFold x);
```

Note that this function fails if neither rule succeeds.

2.4 Choosing between Rules

Rules are first-class values. Therefore, it is also possible to choose between rules. Extending the semantics of choice suggested above, the choice `(`r_1 `<+` r_2`)` between two rules trivially evaluates to r_1 since r_1 is a normal form and does not fail. However, in RhoStratego a choice between two rules is automatically lifted into a rule. That is, in the term `((`t_1 `<+` t_2`)` t_3`)`, the application is distributed over the choice, yielding `(`t_1 t_3 `<+` t_2 t_3`)`. Thus, the choice is made *after* evaluating `(`t_1 t_3`)`. If the application `(`t_1 t_3`)` evaluates to a rule, the process is repeated. That is, the choice is only consumed when the left-hand side of the choice is a constructed value or failure.

Thus, we can reformulate the one-step simplification function by taking the choice between the rules `plusZero` and `plusFold`:

```
foldOne = plusZero <+ plusFold;
```

It is clear that the application-over-choice distribution rule transforms the second definition of `foldOne`, after η-expanding its body, into the first definition.

2.5 Cutting Choice

A choice catches failure anywhere in its left-hand side alternative. This is not always desirable. For example, suppose that we want to distinguish terms representing applications (constructed with `App`) from other terms, and take a different action in each case. The formulation

`App x y ->` t_1 `<+` t_2

is not adequate, because if t_1 fails, after an `App` term has been matched, the alternative t_2 is still evaluated. Thus, after the pattern match succeeds we want to commit to the left alternative of the choice and cut off backtracking to its right alternative. This is achieved using the unary *cut* operator (`^`), which indicates that a function or failure result in the left-hand side of the choice should be left as-is. Using the cut operator the example above can be reformulated as

`App x y ->` `^`t_1 `<+` t_2

Now, after matching an `App` term, the result of t_1 is produced, even if that is failure.

```
plusZero =
  App (App (Prim "+") (Num 0)) x -> x;
plusComm =
  App (App (Prim "+") x) y ->
  App (App (Prim "+") y) x;
plusFold =
  App (App (Prim "+") (Num i)) (Num j) ->
  Num (i + j);
mulFold =
  App (App (Prim "*") (Num i)) (Num j) ->
  Num (i * j);
beta =
  App (Abs x e1) e2 -> Let x e2 e1;
letVar =
  Let x e (Var x) -> e;
letApp =
  Let x e1 (App e2 e3) ->
  App (Let x e1 e2) (Let x e1 e3);
letHoist =
  Let x (Let y e1 e2) e3 ->
  Let y e1 (Let x e2) e3;
appLetL =
  App (Let x e1 e2) e3 -> Let x e1 (App e1 e2);
appLetR =
  App e1 (Let x e2 e3) -> Let x e2 (App e1 e2);
```

Figure 3: Simplification rules

2.6 Creating and Matching Failure

Since failure is a value that can be produced by a term, it is also possible to match against it. Although choice allows catching of failure, explicit matching and creation of failure are needed in a few cases. An application `((fail -> `t_1`)` t_2`)` forces the evaluation of t_2. If the result is failure, the body t_1 is returned, otherwise the application fails. The main applications of matching against failure are strict function application and negation by failure.

Since RhoStratego is a lazy language, the argument of a function is not evaluated before the function is called. Sometimes it necessary to force the evaluation of an argument in order to know whether or not it fails. Strict function application is achieved with the function `st`, defined as:

```
st = f -> ((fail -> ^fail) <+ f);
```

It applies a function strictly to its argument. That is, in the application `st` t_1 t_2 the evaluation of the argument t_2 of the function t_1 is forced by matching it against the pattern `fail`. In case the evaluation of t_2 does not fail the right-

hand side of the choice is taken, and t_1 is applied to the value of t_2. Note that we must write `^fail` in order to prevent `fail` from being caught immediately by the choice.

An application of `st` is the strict sequential composition function `|` ('pipe'), defined as

```
| = f -> g -> t -> st g (f t);
```

which allows us to write transformation pipelines s_1 `|` s_2 `|...|` s_n. Note that this composition operator applies its *left* argument first.

The constant `fail` can be used to create failure outside of a pattern match. This can be used to turn a value into a failure. In the following definition of negation by failure, `fail` is used to force failure when a computation succeeds:

```
neg = f -> x -> (fail -> x <+ fail) (f x)
```

An application `neg` t_1 t_2 returns the value of t_2 if (`(`t_1 t_2`)` evaluates to failure, and fails otherwise.

2.7 Strategy operators

Using this machinery we can define higher-order operators for combining transformations. The combinator `try`

```
try = s -> (s <+ id);
```

tries the application of a transformation `s`, but returns the original term if `s` fails. The fixpoint combinator `repeat`, defined as

```
repeat = s -> try (s | repeat s);
```

applies a transformation `s` repeatedly until it fails. As an application of `repeat` consider the strategy

```
simplifyMany =
  repeat(plusZero <+ (plusComm | plusZero)
         <+ plusFold);
```

which repeatedly applies some simplification rules until none applies anymore.

Note that we cannot use regular sequential composition in the definition of `repeat`, i.e.,

```
repeat = s -> try ((repeat s) . s);
```

as this will cause `repeat` to get stuck in an infinite recursion: `repeat s` expands into `try (repeat s . s)`, which is `(repeat s . s) <+ id`. Since `repeat s . s` is equal to `t -> repeat s (s t)`, and a lazy language first evaluates the left-hand side of this application — namely `repeat s` — we have a loop. We must have some strictness to ensure that progress is made.

3 Generic Traversal

First-class rules and the choice operator enable the specification of a library of transformation rules that can be combined in many ways. In order to apply transformation rules to an abstract syntax tree it is necessary to traverse that tree, i.e., apply rules below the root constructor. Traversal can be accomplished using pattern matching and recursion. However, this induces overhead: all constructors that are traversed must be mentioned explicitly. It is desirable to define traversals generically, using default behavior for most constructors.

Stratego introduced generic traversal primitives such as `all` and `one` that capture schemas for traversing from a constructor application to its arguments [27]. Using a few of such primitives, a wide range of highly generic term traversals can be defined, avoiding the overhead normally associated with the definition of traversals.

In RhoStratego the built-in primitive traversal schemas of Stratego can be defined using one single construct: *application pattern matching*. An application pattern is a pattern of the form `c x`, where `c` and `x` are both variables. An application pattern `c x` matches a constructor application, binding `x` to the last argument of the constructor (the *suffix*), and binding `c` to the constructor applied to the other arguments (the *prefix*).

As an example consider the definition of the generic function `termSize`, which counts the number of constructor nodes (including basic values) in its argument term:

```
termSize =
  c x -> termSize c + termSize x <+ x -> 1;
```

The left alternative is a rule that matches a constructor application and computes the size of the prefix `c` and the suffix `x`. The right alternative matches all terms that are not applications, i.e., constructors and constants. Thus, `termSize (Num 1)` equals `(termSize Num) + (termSize 1)` which equals 2 and `termSize (App (Num 1) (Var "x"))` equals `(termSize (App (Num 1)))` `+ (termSize (Var "x"))`, which eventually evaluates to `5`.

In the rest of this section we consider the implementation of the basic traversals `all`, `one`, and `crush` and their application in the definition of generic traversals.

3.1 All

The `all` operator applies a function `f` to all direct subterms of a term. That is, `all f (C` $t_1 \ldots t_n$`)` evaluates to `C (f` t_1`)...(f` t_n`)`. It can be implemented as follows:

```
all = f -> (c x -> ^(st (all f c) (f x)) <+ id);
```

The left-hand side of the choice matches the argument term against the application pattern `c x`. If the match fails, the argument is either a constructor or

some other value (e.g., an integer) without subterms. In this case the term is left unchanged by applying the identity, `id`. If the match succeeds, the term is of the form `C` $t_1 \ldots t_n$, i.e., a constructor applied to n arguments and `c` is bound to its prefix `C` $t_1 \ldots t_{n-1}$, and `x` is bound to its suffix t_n. The function `f` is applied to the suffix t_n, and, through the recursive call `all f c`, it is applied to the subterms in the prefix, resulting in `C (f` t_1`)...(f` t_{n-1}`)`. Applying the transformed prefix to the transformed suffix completes the construction of the new term `C (f` t_1`)...(f` t_n`)`. Since the function `f` must be *successfully* applied to all subterms, this must be a *strict* application. Finally, the result must be cut in order to prevent failure in `f` from warping us into `id`.

Note that `all f` succeeds on constants, i.e., terms without subterms. Thus, we can define the generic function `isConstant` that succeeds on constants and fails on non-constants as

```
isConstant = all fail;
```

Furthermore, note that `all` forces the evaluation of the arguments of a constructor. Thus `(all id)` has the effect of forcing the evaluation of the direct subterms of a constructor, and `force`, defined as

```
force = all force;
```

forces the complete evaluation of a constructed value.

Using `all`, a wide range of full term traversals can be defined. For example, the function `topdown` defines a full traversal of a term that applies a transformation `s` at every subterm before visiting its children.

```
topdown = s -> s | all (topdown s);
```

Its dual is `bottomup`; it visits the subterms of a term before applying a transformation to it.

```
bottomup = s -> all (bottomup s) | s;
```

An application of `bottomup` is the following constant folding strategy `foldConst` for expressions:

```
foldOne   = plusFold <+ plusZero <+ mulFold;
foldConst = bottomup(repeat(foldOne));
```

The transformation makes a single pass over a term, repeatedly applying folding rules after transforming the subterms of a term.

Finally, the function `alltd` tries to apply a transformation `s` to a term. If that fails it recursively descends into the subterms.

```
alltd = s -> s <+ all(alltd s);
```

This function applies a transformation along an internal frontier of a term, while `topdown` and `bottomup` apply a transformation to all subterms. An application of `alltd` is the substitution function `subst`

```
subst env = alltd(Var x -> lookup x env);
```

which replaces all `(Var x)` subterms by the value of `x` in the environment `env`. If the lookup in the environment fails, a variable is not replaced. Note that no substitution takes place in the expression that a variable is replaced with.

3.2 One

The `one` operator applies a function `f` to exactly one immediate subterm of a term and fails if `f` cannot be applied to at least one such term. Thus, `one f (C` $t_1 \ldots t_n$`)` evaluates to `C` $t_1 \ldots$`(f` t_i`)`$\ldots t_n$, if `f` t_i succeeds and `f` t_j fails for $j > i$. It can be implemented as follows:

```
one = f -> c x -> (st c (f x) <+ one f c x);
```

The left alternative of the choice tries to apply `f` to the suffix `x`. If that succeeds the term is reconstructed with the original prefix and the transformed suffix. Otherwise the right alternative tries to find a subterm in the prefix to apply `f` to using the recursive call `one f c`. If the function hits the constructor, the application match fails and thus no subterm to apply `f` to has been found. For example, the result of `one (A -> B) (C A A B)` is `C A B B`, and the result of `one (A -> B) (C B)` is `fail`. Note that `one` fails for constructors without (direct) subterms. Thus, the definition

```
hasArgs = one id;
```

defines the generic function `hasArgs` that succeeds for terms with subterms.

An example application of `one` is the traversal `oncetd` that searches a term for a subterm to which a transformation `s` can be applied.

```
oncetd = s -> (s <+ one (oncetd s));
```

This traversal fails if `s` cannot be applied to any subterm. An application of `oncetd` is the following simple beta reduction strategy.

```
betaReduce =
  repeat(oncetd(beta <+ appLetL <+ appLetR));
```

Another use of `oncetd` is the function `occurs`, which checks the occurrence of a variable in an expression.

```
occurs = x -> oncetd(Var x -> Var x);
```

3.3 Crush

The traversal primitives `all` and `one` preserve term structure (although traversals such as `topdown` can change the structure of a term). It can also be useful to reduce a constructor application to a value. The `crush` operator is a one-level reduction operator that reduces a constructor application by combining the reductions of the direct subterms in a uniform way. The `crush` operator is implemented as follows:

```
crush = op -> nul -> f ->
  (c x -> op (crush op nul f c) (f x)
  <+ x -> nul);
```

That is, `crush op nul f (C` $t_1 \ldots t_n$`)` evaluates to `(op ... (op (op nul` t_1`) (f` t_2`)) ... (f` t_n`))`. The function `f` is a generic reduction operator and functions `op` and `nul` combine its results.

The generic node counting function `termSize` can be redefined in terms of `crush` as

```
termSize = crush (+) 1 termSize;
```

Note that the function is used recursively to reduce the direct subterms of a term.

A more general application of `crush` is the function `collect`, which collects all outermost subterms on which `s` succeeds.

```
collect = s -> ((s | (y -> [y]))
                <+ crush union [] (collect s));
```

For example, to collect all variables in an expression, `collect` is instantiated with a rule that recognizes variables: `collect(Var x -> x)`. This function produces the set of all variables occurring in an expression. If we want to collect only the *free* variables in an expression we need a refined version of collect in which the user-defined base case is parameterized with the collect algorithm itself:

```
collectR = coll ->
  (let c = collectR coll;
    in coll c <+ crush union [] c);
```

A free variable collection function collects all variables, just as in the instantiation above, but filters out variables bound by the `Abs` and `Let` constructs.

```
freeVars = collectR (fv ->
  (  Var x -> [x]
  <+ Abs x e -> diff (fv e) [x]
  <+ Let x e1 e2 -> union (fv e1)
                          (diff (fv e2) [x])));
```

LetLift: $$e \mapsto \textbf{let in } e$$

LetLet: $$\frac{\text{defs}(ds_1) \cap \text{defs}(ds_2) = \emptyset}{\textbf{let } ds_1 \textbf{ in let } ds_2 \textbf{ in } e \mapsto \textbf{let } ds_1 ds_2 \textbf{ in } e}$$

Var: $$\frac{x{=}e\textbf{;} \in ds}{\textbf{let } ds \textbf{ in } x \mapsto \textbf{let } ds \textbf{ in } e}$$

Beta: $$\frac{x \notin \text{defs}(ds)}{\textbf{let } ds \textbf{ in } (x \ \rightarrow \ e_1)\ e_2 \mapsto \textbf{let } ds \textbf{ in let } x{=}e_2\textbf{; in } e_1}$$

ConMatch$^+$: $$\textbf{let } ds \textbf{ in } (C \ \rightarrow \ e)\ C \mapsto \textbf{let } ds \textbf{ in } e$$

ConMatch$^-$: $$\frac{C_1 \neq C_2}{\textbf{let } ds \textbf{ in } (C_1 \ \rightarrow \ e)\ C_2 \mapsto \textbf{let } ds \textbf{ in fail}}$$

AppMatch$^+$: $$\frac{(e_1\ e_2) \text{ is a normal form}}{\begin{array}{c}\textbf{let } ds \textbf{ in } (p_1\ p_2 \ \rightarrow \ e_3)\ (e_1\ e_2) \mapsto \\ \textbf{let } ds \textbf{ in } (p_1 \ \rightarrow \ p_2 \rightarrow \ e_3)\ e_1\ e_2\end{array}}$$

AppMatch$^-$: $$\frac{e_1 \text{ is a normal form but not an application}}{\textbf{let } ds \textbf{ in } (p_1\ p_2 \ \rightarrow \ e_3)\ e_1 \mapsto \textbf{let } ds \textbf{ in fail}}$$

FailMatch$^+$: $$\textbf{let } ds \textbf{ in } (\textbf{fail} \ \rightarrow \ e)\ \textbf{fail} \mapsto \textbf{let } ds \textbf{ in } e$$

FailMatch$^-$: $$\frac{e_2 \text{ is a normal form } \wedge \ e_2 \neq \textbf{fail}}{\textbf{let } ds \textbf{ in } (\textbf{fail} \ \rightarrow \ e_1)\ e_2 \mapsto \textbf{let } ds \textbf{ in fail}}$$

Figure 4: RhoStratego evaluation rules

Thus, a specialized collection algorithm is defined that only needs to mention the constructors that are relevant for the problem at hand. All other constructors are handled by the generic default case.

4 Semantics

In this section we present the semantics of the RhoStratego language as a set of rewrite rules on the language, together with a lazy evaluation strategy for reducing terms to normal form. Integer and string constants have been omitted for brevity.

4.1 Evaluation Rules

The rewrite rules defining the semantics of RhoStratego are given in Figure 4. We write $e_1 \mapsto e_2$ to denote that there is a sequence of rewrite steps that transforms e_1 into e_2. A let-expression is in *normal form* if no rules apply;

$$\textsc{EvalFunc}: \quad \frac{\textbf{let } ds \textbf{ in } e_1 \mapsto \textbf{let } ds' \textbf{ in } e_1'}{\textbf{let } ds \textbf{ in } e_1\ e_2 \mapsto \textbf{let } ds' \textbf{ in } e_1'\ e_2}$$

$$\textsc{EvalArg}: \quad \frac{\textbf{let } ds \textbf{ in } e_2 \mapsto \textbf{let } ds' \textbf{ in } e_2'}{\textbf{let } ds \textbf{ in } e_1\ e_2 \mapsto \textbf{let } ds' \textbf{ in } e_1\ e_2'}$$

$$\textsc{PropFunc}: \quad \textbf{let } ds \textbf{ in fail } e \mapsto \textbf{let } ds \textbf{ in fail}$$

$$\textsc{EvalLeft}: \quad \frac{\textbf{let } ds \textbf{ in } e_1 \mapsto \textbf{let } ds' \textbf{ in } e_1'}{\textbf{let } ds \textbf{ in } e_1 \texttt{ <+ } e_2 \mapsto \textbf{let } ds' \textbf{ in } e_1' \texttt{ <+ } e_2}$$

$$\textsc{LChoice}: \quad \frac{\begin{array}{c}e_1 \text{ is a normal form } \wedge\ e_1 \neq \textbf{fail} \wedge \\ e_1 \text{ is not a cut or a function}\end{array}}{\textbf{let } ds \textbf{ in } e_1 \texttt{ <+ } e_2 \mapsto \textbf{let } ds \textbf{ in } e_1}$$

$$\textsc{LChoiceCut}: \quad \textbf{let } ds \textbf{ in } \texttt{\^{}}e_1 \texttt{ <+ } e_2 \mapsto \textbf{let } ds \textbf{ in } e_1$$

$$\textsc{RChoice}: \quad \textbf{let } ds \textbf{ in fail <+ } e \mapsto \textbf{let } ds \textbf{ in } e$$

$$\textsc{UncutFunc}: \quad \textbf{let } ds \textbf{ in } \texttt{\^{}}e_1\ e_2 \mapsto \textbf{let } ds \textbf{ in } e_1\ e_2$$

$$\textsc{UncutArg}: \quad \frac{p \text{ is a strict pattern}}{\textbf{let } ds \textbf{ in } (p \to e_1)\ \texttt{\^{}}e_2 \mapsto \textbf{let } ds \textbf{ in } (p \to e_1)\ e_2}$$

$$\textsc{Distrib}: \quad \textbf{let } ds \textbf{ in } (e_1 \texttt{ <+ } e_2)\ e_3 \mapsto \textbf{let } ds \textbf{ in } e_1\ e_3 \texttt{ <+ } e_2\ e_3$$

Figure 4: Continued

that is, if its body is a rule, a constructor applied to zero or more (possibly unnormalised) arguments, a failure, a cut, or a choice, if the left-hand side of the choice is a rule.

It is assumed that the left-hand side term has the form **let** ds **in** e. The idea is that the let-environment represents the memory, or heap, of the abstract machine. This allows us to express certain aspects of the operational semantics, such as garbage collection and sharing. Since not all RhoStratego terms are `let`s, we have the trivial rule LetLift to lift these into the canonical form. Note that a RhoStratego program is a set of declarations; declarations are variable definitions, data type declarations, and type signatures (the latter two not being discussed here). The semantics of a whole program is obtained by lifting the set of declarations ds into a **let** and evaluating the variable **main**, i.e., **let** ds **in main**.

Lets If the body of a let is a let, we can merge the definitions, provided that there are no name clashes (the LetLet rule). This rule can be interpreted as

allocating closures for the values defined in the let-expression. We implicitly assume that α-renaming takes place as required.

Variables The VAR rule expresses that a variable may be substituted by its definition. As stated above, we can use the let-environment to express aspects of the operational semantics. Here is an alternative VAR rule:

$$\textsc{Var}' : \frac{x{=}e; \in ds \ \wedge \ \textbf{let } ds \textbf{ in } e \mapsto \textbf{let } ds' \textbf{ in } e'}{\textbf{let } ds \textbf{ in } x \mapsto \textbf{let } ds' \wr (x,\ e') \textbf{ in } e'}$$

where $ds' \wr (x,\ e')$ denotes ds' with the definition for x replaced by $x = e';$. The idea is that a variable is evaluated (presumably to normal form), and the result is written back into the 'heap' (removing the old definition for x). Then, if x is needed again, we do not need to evaluate it again; it is already done. So the alternative VAR rule nicely captures the operational notion of *sharing*; it corresponds with the implementation technique of preventing work duplication by updating a closure with its result.

Applications The BETA rule expresses the fundamental axiom of the λ-calculus, β-reduction, by means of *explicit substitution* [1]: rather than having a substitution operation, substitutions are expressed in the language itself. We do this by adding the argument to the let-environment, and then evaluating the body of the rule. All initial terms are assumed to be *closed*, i.e., contain no free variables. As a consequence there is no need to add the restriction that x should not occur free in e_2, since the fact that it does not occur in ds implies it cannot occur free in e_2.

We can evaluate the left and right sides of a function application using the EVALFUNC and EVALARG rules. EVALARG is necessary in strict pattern matches, i.e., matches against constructors, applications, or failure (CONMATCH$^+$ to FAILMATCH$^-$). Note that for an application pattern match to succeed, the argument should be in normal form and an application. This implies that it is a constructed value. Applying failure to an expression yields failure (PROPFUNC).

From the definition of APPMATCH$^+$ the exact semantics of the application pattern match follows. For example, `(c x -> c) (A B C)` evaluates to `A B`, and `(c x -> x) (A B C)` evaluates to `C`. In essence, it allows us to look at the immediate subterms of a term in a linear fashion, just like traversing a `Cons`/`Nil` list. The `c x` pattern corresponds to matching a `Cons`, and the pattern `x` (anything else) corresponds to matching with `Nil`. That is, if the match `c x` fails, the argument is either a constructor (without arguments) or another normal form (such as a function or an integer literal).

Choices The remaining rules deal with the evaluation of choices. We evaluate choices by first evaluating the left alternative (using EVALLEFT). We can choose using LCHOICE the left alternative if it is not a failure, a rule or a cut; this implies that it should have been evaluated to normal form, since otherwise

we cannot know that it is not a failure. If it is a cut, we can choose the left alternative using LChoiceCut which removes the cut. Note that only one cut is removed; this allows an expression to escape several choices by applying several cuts. If it is a function, we can use the Distrib rule to distribute arguments over the alternatives. If it is a failure, we can choose the right alternative using RChoice.

Finally, cuts not occurring as the left argument of a choice 'disappear' (the UncutFunc and UncutArg rules; a strict pattern is a pattern that forces evaluation of the argument, i.e., anything other than a variable). This is to ensure that for example `(^id) C` or `(C -> D) ^C` works. This makes it easier to reuse functions returning cuts.

The following alternative Distrib rule is preferable from an operational point of view, since it is more efficient:

$$\textsc{Distrib}: \frac{x \notin \text{defs}(ds)}{\begin{array}{l}\textbf{let } ds \textbf{ in } (e_1 \texttt{ <+ } e_2)\ e_3 \mapsto \\ \textbf{let } ds \textbf{ in let } x{=}e_3\textbf{; in } e_1\ x \texttt{ <+ } e_2\ x\end{array}}$$

Together with the alternative Var rule, this prevents e_3 from being evaluated more than once.

4.2 Evaluation Strategy

The rules discussed above define one-step reductions of terms. A complete evaluation of a term requires repeated application of rules. Depending on the strategy that is chosen, different effects can be achieved. Bot lazy and strict interpretations can be achieved using the rules. Lazy and strict evaluation based on the same set of rules has been implemented in Stratego and is presented in [13]. Below we give an informal account of the lazy strategy that is the basis for the compiler.

Reducing a term to normal form involves applying the rules in Figure 4 to the term to be reduced. We first need to make precise what applying a rule to a term means. For simple rules such as Beta or ConMatch$^+$ that have no or only simple conditions, this is unambiguous: we can apply the rules if the conditions are satisfied. However, the *evaluation rules* (e.g. EvalFunc) are conditional upon some term e_1 being rewritable into e_2. This means that such a rule must be parameterised with some strategy that reduces e_1.

The strategy E evaluates a term e lazily as follows. If e is in normal form, we are done. Otherwise, we must apply one or more rules. We must be careful that we always make some progress; for example, the EvalLeft rule is always applicable if e is an application. The LetLet or Var are always safe to apply.

If e is a choice, we must first apply the EvalLeft rule with strategy E to normalise the left-hand side of the choice. It is vital that we now make some more progress, in order to prevent infinite loops (since EvalLeft will continue to be applicable): we have to get rid of the choice. We do this by applying the

LChoice, LChoiceCut, or RChoice rules; exactly one should be applicable.

If none of the above applied, we are dealing with an application. In a lazy semantics we have to evaluate the left-hand side first. This means that we have to get rid of any cuts, so we first apply the UncutFunc rule until it becomes inapplicable. Then we can apply the EvalFunc rule with strategy E to normalise the left-hand side. Just as with choices, we must apply some other rule next to get rid of the application, *unless* the application is a constructor application (`C` e_1 ... e_n), which is a normal form. We can now try one of the Beta, Distrib, or PropFunc rules; otherwise, we are looking at a strict pattern match: a match against a constructor, application, or failure. This requires that we remove cuts from the argument, so UncutArg should be applied until it becomes inapplicable. Then we can apply the EvalArg rule with strategy E to normalise the argument, followed by one of the ConMatch$^+$ etc. rules to perform the actual reduction.

Afterwards, we can apply the strategy again (i.e., iteratively) to complete the evaluation of e.

5 Type System

In this section we describe a type system for RhoStratego, which is based on the Hindley-Milner type system [19]. The main issue is how to type generic traversals. To this end, we add rank-2 types and rules for typing application pattern matches and generic traversals.

5.1 Data Type Declarations

Constructors are defined separately from the data types they construct. This allows data types to be extended, possibly in separately compiled modules. For example, lists can be defined as follows:

```
data List a;
Nil :: List a;
Cons :: a -> List a -> List a;
```

5.2 Application Pattern Matches

Generic traversal functions such as `all` and `one` are implemented using application pattern matches. How should this this language construct be typed?

We should first consider what the intended type of a function such as `all` is. We can then try to find typing rules to obtain the desired types. `All` applies a function to all subterms of a term. Since the subterms can have any type, the type of the function should be $\forall\alpha.\alpha \to \alpha$. Therefore `all` should have type $\forall\beta.(\forall\alpha.\alpha \to \alpha) \to \beta \to \beta$. This is a rank-2 type and hence not supported by the Hindley-Milner type system, which universally quantifies type variables only at

the 'top' of a type. As a result RhoStratego needs a rank-n (where $n \geq 2$) type system.

Recall the definition of `all`:

```
all = f -> (c x -> ^(st (all f c) (f x)) <+ id);
```

The type of `id` after instantiation is of course $\tau_0 \to \tau_0$ (we use τ_i to denote fresh type variables). The left and right arguments to a choice must have the same type. The left-hand choice, `c x` $\to$ `^(st (all f c) (f x))`, must therefore also have this type.

In order to derive the type of this expression, we must assign types to the variables `c` and `x`. Since the pattern match only succeeds for a pattern match against a constructed value of some type τ_1, `c` must be a constructor function which expects an argument of some type τ_2 and returns a value of type τ_1, i.e., `c` has type $\tau_2 \to \tau_1$ and `x` has type τ_2. Hence, the type of `c x` is τ_1. (Things are actually a bit more complicated than that; the exact typing rules for application pattern matches are given below).

Now we can assign a type to the body of the rule, `^(st (all f c) (f x))`. Cuts are irrelevant to the type and `st` (strict application) has the uninteresting type $\forall\alpha.\forall\beta.(\alpha \to \beta) \to \alpha \to \beta$, so we are left with `all f c (f x)`. Assuming that we have the following type judgments:

`all`	::	$\forall\beta.(\forall\alpha.\alpha \to \alpha) \to \beta \to \beta$
`f`	::	$\forall\alpha.\alpha \to \alpha$
`c`	::	$\tau_2 \to \tau_1$
`x`	::	τ_2

we can derive:

`all f`	::	$\tau_3 \to \tau_3$	(after instant. $\forall\beta$ with τ_3)
`all f c`	::	$\tau_2 \to \tau_1$	(i.e., $\tau_3 ::= \tau_2 \to \tau_1$)
`f x`	::	τ_2	(after instant. $\forall\alpha$)
`all f c (f x)`	::	τ_1	

We conclude that the type of the left choice argument is $\tau_1 \to \tau_1$, which matches neatly with the right argument.

5.3 Genericity

We have now seen that `all` can be typed, and that it has the type $\forall\beta.(\forall\alpha.\alpha \to \alpha) \to \beta \to \beta$, and so applies a function of type $\forall\alpha.\alpha \to \alpha$ to the subterms. Unfortunately, in most pure languages essentially the only function with that type is `id` (i.e., $\lambda x.x$). Hence, we cannot write the following:

```
rename = all(try(Var "x" -> Var "y"));
```

which attempts to rename direct subterms, since the argument to `all` has type `Exp` $\to$ `Exp` which is not polymorphic. Therefore RhoStratego provides a *run-time type check mechanism*. We write:

```
rename = all(try(Exp?Var "x" -> Var "y"));
```

The meaning of a pattern $t?p$ (where t is a type and p a pattern) is that the argument is first checked — at runtime — to be of type t. If it is, we proceed as usual, matching against p. Otherwise, the result is `fail`.

We type a pattern $t?p$ by adding *guarded types*. A guarded type is a type prefixed by a question mark, e.g., `?Exp`. We also define the type of a runtime type check pattern $t?p$ to be $?t$. The type of `Exp?Var "x" → Var "y"` therefore is `?Exp → Exp` (i.e., `(?Exp) → Exp`). The trick is that a type match between a function type $\alpha \to \tau_1$ and $?\tau_2 \to \tau_3$ is performed by matching $\alpha \to \tau_1$ against $\tau_2 \to \tau_3$, but all substitutions found for α are filtered out and are not applied to the type environment. The rationale is that a function with a runtime type check pattern really does match anything (i.e., the pattern $?\tau_2$ should match with α without a substitution $\alpha := \tau_2$ taking place), but since the body of the function is only reached when the argument is of type τ_2, the type of the body τ_3 is irrelevant when the argument is *not* of type τ_2, and we can just pretend that it is polymorphic.

In the example above, then, we can successfully match $\alpha \to \alpha$ against `?Exp → Exp` (with the substitution $\alpha :=$ `Exp` filtered out), which can be generalized so that it is a valid argument to `all`. On the other hand $\alpha \to \alpha$ does not match against `?Exp → String`.

The function `all` is an example of a *type preserving* function, in which the type of the output is the same as the type of the input. We also encounter *type unifying* functions, which map everything to the same type. An example is `collect`, which has type $\forall\alpha.\forall\beta.(\forall\gamma.\gamma \to \beta) \to \alpha \to [\beta]$ (the argument function maps everything to β). For example:

```
varNames = collect (Exp?Var x -> x);
```

The type of `Var x -> x`, `?Exp → String`, is an instance of $\forall\gamma.\gamma \to \beta$, with `String` substituted for β.

The RhoStratego type system restricts guarded types to constructed types only; they cannot be functions. Furthermore, the types must be general (`[Int]` is not allowed; `[a]` is). The reason for this is that we do not want to carry runtime type information for all values. For constructed values, this information must be carried around in any case, since we must be able to distinguish between constructors.

5.4 Typing rules

The inference rules of the type system are given in Figures 5 and 6, for terms and patterns respectively (the rule Abs requires that we can assign a type to a pattern).

Type assignments for constructor functions are part of the environment Γ, as expressed by Con and PCon rules.

$$\textsc{Var}: \quad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\textsc{Con}: \quad \frac{C : \tau \in \Gamma}{\Gamma \vdash C : \tau}$$

$$\textsc{Fail}: \quad \Gamma \vdash \texttt{fail} : \tau$$

$$\textsc{Cut}: \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \hat{}e : \tau}$$

$$\textsc{Abs}: \quad \frac{\Gamma' = \Gamma \cup \Gamma_p \ \wedge \ \Gamma' \vdash p : \sigma \ \wedge \ \Gamma' \vdash e : \tau}{\Gamma \vdash (p \to e) : \sigma \to \tau}$$

$$\textsc{App}: \quad \frac{\Gamma \vdash e_1 : \sigma \to \tau \ \wedge \ \Gamma \vdash e_2 : \sigma}{\Gamma \vdash (e_1 e_2) : \tau}$$

$$\textsc{Inst}: \quad \frac{\Gamma \vdash e : \forall \alpha.\tau}{\Gamma \vdash e : [\alpha := \sigma]\tau}$$

$$\textsc{Gen}: \quad \frac{\Gamma \vdash e : \tau \ \wedge \ \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\tau}$$

$$\textsc{Let}: \quad \frac{\begin{array}{l}\Gamma' = \Gamma \cup \Gamma_{ds} \ \wedge \ \Gamma' \vdash e : \tau \ \wedge \\ \forall (x = e'\mathbf{;}) \in ds : (\Gamma' \vdash e_2 : \tau_2 \ \wedge \ x : \tau_2 \in \Gamma')\end{array}}{\Gamma \vdash (\mathbf{let}\ ds\ \mathbf{in}\ e) : \tau}$$

$$\textsc{Choice}: \quad \frac{\Gamma \vdash e_1 : \tau \wedge \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \texttt{<+} e_2 : \tau}$$

$$\textsc{Widen}: \quad \frac{\Gamma \vdash e : \mathbf{?}\sigma \to ([\alpha := \sigma]\tau)}{\Gamma \vdash e : \alpha \to \tau}$$

$$\textsc{Contract}: \quad \frac{\begin{array}{l}\Gamma \vdash e : \tau[\mathbf{Gen}(\alpha_0, \alpha_1, \ldots, \alpha_n)] \wedge \\ (\forall i, 0 \leq i \leq n : \alpha_i \notin \mathrm{fv}(\Gamma)) \wedge \\ (\forall i, 1 \leq i \leq n : \alpha_i \notin \mathrm{fv}(\tau))\end{array}}{\Gamma \vdash e : \tau[\alpha_0]}$$

Figure 5: Typing rules for terms

$$\textsc{PVar}: \quad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\textsc{PCon}: \quad \frac{n \geq 0 \ \wedge \ C : (\sigma_1 \to \ldots \to \sigma_n \to \tau) \in \Gamma \ \wedge \ \Gamma \vdash p_1 : \sigma_1 \ \wedge \ \ldots \ \wedge \ \Gamma \vdash p_n : \sigma_n}{\Gamma \vdash (C \ p_1 \ \ldots \ p_n) : \tau}$$

$$\textsc{Rttc}: \quad \frac{\Gamma \vdash p : \sigma}{\Gamma \vdash (\sigma \mathbf{?} p) : \mathbf{?}\sigma}$$

$$\textsc{Generic}: \quad \frac{n \geq 1 \ \wedge \ x_0 : (\alpha_1 \to \ldots \to \alpha_n \to \alpha_0) \in \Gamma \ \wedge \ x_1 : \alpha_1 \in \Gamma \ldots \ \wedge \ x_n : \alpha_n \in \Gamma}{\Gamma \vdash (x_0 \ x_1 \ \ldots \ x_n) : \mathbf{Gen}(\alpha_0, \alpha_1, \ldots, \alpha_n)}$$

Figure 6: Typing rules for patterns

In the Abs and Let rules, Γ_p and Γ_{ds} refer to an environment that contains type assignments for all variables defined in the pattern or let-binding, respectively.

The Rttc pattern typing rule (for 'runtime type check') introduces guarded types. They can be eliminated through the Widen rule. For example, the type `?Id` $\to$ `String` can be 'widened' to $\alpha \to$ `String`, and `?Id` $\to$ `Id` can be widened to $\alpha \to \alpha$.

The Generic rule assigns a type to an application pattern match $x_0 \ x_1 \ \ldots \ x_n$. We assign $x_1 \ldots x_n$ types $\alpha_1 \ldots \alpha_n$, being fresh type variables. Then x_0 has type $\alpha_1 \to \ldots \to \alpha_n \to \alpha_0$, where α_0 is a fresh type variable. In principle, the type of the entire pattern is α_0. However, we must restrict the substitutions that can be made against these type variables; otherwise, the result will not be generic. For example, consider the following definitions:

```
prefix = c x -> c;
suffix = c x -> x;
f = c x -> c (x + 1);
```

If application pattern matches are implemented naively, then `prefix` will have type $\forall\alpha.\forall\beta.\alpha \to (\beta \to \alpha)$, `suffix` will have type $\forall\alpha.\forall\beta.\alpha \to \beta$, and `f` will have type $\forall\alpha.\alpha \to \alpha$. All those types are too general.

We solve this problem by assigning $x_0 \ x_1 \ \ldots \ x_n$ the type $\mathbf{Gen}(\alpha_0, \alpha_1, \ldots, \alpha_n)$. **Gen** is an intermediate construct that must eventually be *contracted* into α_0 if and only if the αs do not occur in the environment and $\alpha_1 \ldots \alpha_n$ do not occur free in the rest of the type. This is expressed by the Contract rule. This, and the fact that the αs are variables, implies that no constraints to the resulting type can be found later on, and that therefore genericity is ensured. We use the notation $\tau[\sigma]$ to refer to a hole in τ.

6 Implementation

The implementation of RhoStratego consists of a parser, type inferencer, interpreter, compiler and standard library.

The interpreter is written in Stratego and based directly on the semantic rewrite rules given in Section 4. Using the same set of rules, but different strategies a lazy and a strict semantics are provided.

The compiler is also implemented in Stratego and translates RhoStratego programs to C. C's `setjmp/longjmp` exception handling mechanism is used to implement failure and choices. The compiler supports separate compilation of RhoStratego modules.

The compiler and runtime system support the full untyped semantics of the language. This implies that constructors can be applied to an arbitrary number of arguments of any type. We implement this by means of a binary application constructor. For example, the application `C 1 2` is stored in the runtime system as `App (App C 1) 2`. This representation is somewhat wasteful in terms of memory and time usage. However, it makes the efficient implementation of application pattern matches very easy: a pattern `c x` binds `c` and `x` to the left and right arguments of the application constructor, respectively. Note that if all arguments to a constructor were to be stored in the same memory object, an application pattern match would need to make a copy of its argument due to sharing issues.

A type inferencer based on the typing rules in the previous section has also been implemented and integrated with the compiler (the details of the inference algorithm are given in [13]). All types are inferred automatically, except that the programmer must provide type signatures for functions with rank-2 types. A program that passes the type checker is guaranteed not to create non-wellformed terms.

The standard library of RhoStratego contains a range of generic functions including the ones discussed in Sections 2 and 3. An important feature of the library is the ability to read and write *ATerms* [6], a standardised representation for terms used by Stratego and the XT bundle of program transformation tools [11], amongst others. This ensures easy integration of tools such as parsers, pretty-printers, and Stratego specifications.

7 Related Work

7.1 Extensions to Pattern Matching

In this section we take a closer look at the choice operator, and its impact on programming in RhoStratego. It is well known that conventional pattern matching is not perfect [25, 14]. It turns out that the choice operator eliminates the need for many of the proposals to extend pattern matching in functional languages such as Haskell, including *views*, *pattern guards*, *transformational*

patterns, and *first-class patterns*. Furthermore, it makes `case`-expressions and sugar such as Haskell's 'equational style' unnecessary.

Case Expressions Case expressions, present in most functional languages, are unnecessary in the presence of the choice operator, and consequently they do not exist in RhoStratego. After all, the following Haskell-like construct

```
f = x -> case x of
           A       -> 123;
           B "foo" -> 456;
           y       -> 0;
```

can easily be written in RhoStratego as a set of choices. In fact, the choice alternatives are first-class and can be defined separately:

```
a = A       -> 123;
b = B "foo" -> 456;
c = y       -> 0;
f = a <+ b <+ c;
```

Similarly, Haskell's equational style is unnecessary. In the equational style a function definition consists of a number of pattern-guarded equations which must be tried one after another. The various definitions for a function can be written as choice alternatives. The resulting code is in fact shorter than when written in the equational style.

Views Views [28] were proposed to address the problem that regular pattern matching is rather limited since we can only match with actual constructors. As a consequence we cannot match against, e.g., the end of a list instead of the head, nor can we match against abstract data types since there is simply nothing to match against. Using the views proposal for Haskell [8] we can write the following view to match against the end of a list:

```
view Tsil a of [a] = Lin | Snoc y ys where
  tsil xs =
    case reverse xs of
      [] -> Lin
      (y:ys) -> Snoc y ys
```

where matching against a `Snoc`-constructor causes the function `tsil` to be applied to the value:

```
f (Snoc y ys) = y
f Lin = 0
```

It is worth pointing out why views (and transformational patterns) are useful. The reason is that the equational style can only be used if the non-applicability

of an equation can be discovered in the pattern. When that is not possible, the equational style falls apart, and we have to explicitly write the traversal through the alternatives (the equations) as a series of ever more deeply nested `case`-expressions. The choice operator liberates us from this regime, hence the main motivation for views and transformational patterns disappears. With the choice operator, the previous example becomes:

```
f = reverse | ((y:ys) -> y <+ [] -> 0);
```

Views still have the advantage that the transformation to be applied (e.g., `tsil`) is implicit in the name of the patterns (e.g., `Snoc`), but this seems only a minor advantage.

Pattern Guards In Haskell's equational notation, we can use boolean guards to further restrict the applicability of an equation, e.g., `f x | x > 3 = 123` (the symbol `|` should not be confused with RhoStratego's sequential composition operator). However, there is a disparity between patterns and guards: patterns can bind variables, whereas guards cannot. For example, if we want to return a variable from an environment, or 0 if it is undefined, we would write:

```
f env var | isJust (lookup env var)
          = fromJust (lookup env var)
f env var = 0
```

where `lookup` has type `[(`α`, `β`)]` $\rightarrow$ α $\rightarrow$ `Maybe` β. This is awkward because we now inspect the result of `lookup` twice. Pattern guards [14] redefine a guard as a list of qualifiers, just as in a list comprehension, so that binding can occur:

```
f env var | Just x <- lookup env var = x
f env var = 0
```

But when we have a choice operator, we can simply write the above as a choice. In fact, we could just get rid of the `Maybe` result of `lookup` altogether, making it of type `[(`α`, `β`)]` $\rightarrow$ α $\rightarrow$ β, and arrive at:

```
f = env -> var -> (lookup env var <+ 0);
```

Transformational Patterns Transformational patterns [14] provide a cheap alternative to views, allowing us to write the previous example as:

```
f env (Just x)!(lookup env) = x
f env var = 0
```

Hence, transformational patterns are just view transformations made explicit. The choice operator allows a similar notation:

```
f = env -> ((lookup env) | (Just x -> x)
            <+ y -> 0);
```

However, it is still desirable to have some mechanism like views or transformational patterns, if only for reasons of symmetry. It is ugly if patterns can only be used to match against concrete data types. For this reason RhoStratego offers a syntactic sugar similar to, but slightly simpler than, transformational patterns. The snoc-list example given above can be written in RhoStratego as follows:

```
snoc = reverse | ((x:xs) -> <x, xs>);
lin = [] -> <>;
f = {snoc} x xs -> x <+ {lin} -> 0;
```

The angle brackets denote tuple construction. A pattern `{...}` with n pattern arguments specifies an expression which is applied to the argument. The expression should return a tuple of arity n. The pattern arguments are then matched against the elements of the tuple. Therefore, `f` is desugared into:

```
f =  y -> (<x, xs> -> x) (snoc y)
  <+ y -> (<>      -> 0) (lin y);
```

First-class Patterns Tullsen [25] treats patterns as functions of type $\alpha \rightarrow$ `Maybe` β, and combinators are provided to combine basic patterns into complex ones. Although conceptually elegant, this approach suffers from the fact that the syntax is not very attractive. Furthermore, every function that can fail must have the `Maybe` type; in our approach, failure is propagated implicitly.

7.2 ρ-calculus

RhoStratego's name derives from its origin as an experimental implementation of the *ρ-calculus*. The ρ-calculus, or *rewriting calculus* [9] aims to integrate first-order rewriting, the λ-calculus, and non-determinism. Although we have diverged from that goal and RhoStratego has become a conventional functional language extended with first-class rules and generic traversals, it is still interesting to compare RhoStratego to the ρ-calculus.

The syntax of the ρ-calculus is defined as follows:

$$t ::= \mathit{var} \mid \mathit{constant} \mid t \rightarrow t \mid t \bullet t \mid \mathit{null} \mid t, t$$

$t \rightarrow t$ represents abstraction, $t \bullet t$ stands for application, t, t builds a structure, and *null* denotes the empty structure. It is easy to see that the λ-calculus and term rewriting can be encoded in the ρ-calculus. For example, $x \rightarrow y \rightarrow x$ is exactly the λ-term $\lambda x.\lambda y.x$, and $(F(x) \rightarrow x) \bullet F(A)$ is the rewrite rule $F(x) \rightarrow x$ applied to the term $F(A)$.

The principal semantic rule of the calculus is the FIRE rule, which is essentially a generalized form of β-reduction:

$$(t_1 \rightarrow t_2) \bullet t_3 \mapsto \begin{cases} \mathit{null} & \text{if } \mathrm{Sol}(t_1 \ll_{\mathbb{T}} t_3) = \emptyset \\ \sigma_1 t_2, \ldots, \sigma_n t_2 & \text{where } \sigma_i \in \mathrm{Sol}(t_1 \ll_{\mathbb{T}} t_3) \end{cases}$$

The matching theory $\mathbb{T}$, which is a parameter of the calculus, determines the solutions and substitutions arising out of a match ($\mathrm{Sol}(t_1 \ll_{\mathbb{T}} t_3)$ returns the set of substitutions). In addition, the calculus has the *distribution rules* $(t_1, t_2) \bullet t_3 \mapsto t_1 \bullet t_3, t_2 \bullet t_3$ and $\mathit{null} \bullet t \mapsto \mathit{null}$.

If we view the structure-building operator (,) as a choice operator and *null* as failure, then the distribution rules correspond to the DISTRIB and PROPFUNC rules of RhoStratego. Furthermore, if we take as the matching theory $\mathbb{T}$ the theory of equality modulo α-renaming, then the FIRE rule corresponds to our choice rules, except that there are no cuts. A rule either succeeds and has exactly one solution, or it fails and yields failure. Finally, the proper choice semantics is obtained by defining $\mathbb{T}$ such that t_1, t_2 is equivalent to t_1 if t_1 is not *null*, or to t_2 otherwise.

7.3 Polytypic Programming

Generic or polytypic programming makes it possible to write functions that operate on different data types. For instance, functions such as `termSize` can be readily defined in PolyP [16] or using derivable type classes [15].

However, other kinds of generic traversals are more troublesome. Consider `topdown (Exp?Var "x" -> Var "y")` (i.e., rename a variable named `"x"` to `"y"`). We *can* write generic code for this using derivable type classes, for example. We create a class together with a generic default method to traverse over arbitrary data types, and declare all data types we want to traverse over to be instances of this class, except that we write a *specific* instance for type `Exp` to actually perform the renaming.

The problem with this approach is that there can be at most one class instance declaration for each type, so if we want to define another topdown traversal, we have to define a *new* class, providing different instances for the appropriate types. And we cannot write a general class `Topdown` containing a method that is parameterized with a function applying the transformation, since such a function would necessarily have type $\alpha \to \alpha$ (it must operate on all types), i.e., we can only pass the function `id`.

The idea behind type-safe functional strategies [18] is similar: genericity is achieved through the use of Haskell type classes (which may be generated automatically using an external tool).

8 Future Work

There are several ideas for further development. Stratego has been used to build transformation systems that process real programs (e.g., the Stratego compiler, a Cobol renovation system, a C++ transformation system [3], grammar engineering tools). We still have to test the scalability of RhoStratego. For performance reasons, it would be interesting to retarget the compiler to use an

existing high-quality back-end. C--[22] is a possibility, since it has substantial support for exceptions [24].

Better control over the scope of the choice operator is desirable, i.e., more powerful exception handling primitives. One way to do this is to have several kinds of exceptions, but this destroys the confluence of the language unless the evaluation order is fixed [23]. This is because different subexpressions may raise different exceptions, and so the exception that is actually raised depends on which subexpression is evaluated first.

There is a tension between rewriting and laziness. A term `C fail <+ ...` will always succeed and never go to the right-hand side choice, since `fail` is evaluated lazily. Since in rewriting terms are usually consumed (and therefore produced) in their entirety, a solution is to keep the language lazy, but make constructor application strict. This means that we lose the ability to make infinite or cyclic data structures, but we keep the other advantages of laziness (e.g., the ability to define control structures).

It would be interesting to explore several other Stratego features in a functional context, such as dynamic rules [26] and an explicit match operator. The latter would give us first-class patterns, i.e., the ability to name and abstract over patterns. We are also interested in *non-local variable bindings*. For example, we can write `{x: fetch(?Foo(x)); !x}` in Stratego. That is, walk over a list until an element is encountered that matches with the pattern `Foo(x)`; then replace the term with `x`. Note that `x` is declared in the scope of the entire strategy, but it is defined (given a value) as a side-effect of executing `fetch`. It is not clear how to implement this cleanly in a pure functional language.

9 Conclusion

We have presented the design of a lazy functional language integrating features from the paradigm of rewriting strategies.

Application pattern matches are a simple but quite powerful primitive for constructing generic traversals. They can be typed using the Generic and Contract typing rules, and be put to use with the runtime type check mechanism embodied in the Rttc and Widen rules. Together they allow type unifying and type preserving functions to be written safely.

The choice operator liberates pattern matching from the harness of the case construct. The combination of rules and choice subsumes language constructs such as cases, the equational style, views, pattern guards, and transformational patterns. The cut operator allows fine control over the backtracking behavior of the choice operator.

The implementation of RhoStratego is available under the terms of the GNU General Public License for experimentation at `http://www.stratego-language.org/rho/`.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *17th ACM Symp. on Principles of Programming Languages (POPL'90)*, pages 31–46, San Francisco, California, January 1990. 4.1

[2] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. 1

[3] O. S. Bagge, M. Haveraaen, and E. Visser. CodeBoost: A framework for the transformation of C++ programs. Technical Report UU-CS-2001-32, Institute of Information and Computing Sciences, Utrecht University, 2001. [ps.gz] . 8

[4] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. Elan: A logical framework based on computational systems. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 1996. Proceedings of the First Workshop on Rewriting Logic and Applications 1996 (WRLA'96). 1, 1

[5] J. M. Boyle, T. J. Harmer, and V. L. Winter. The TAMPR program transformation system: Simplifying the development of numerical software. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 353–372. Birkhäuser, 1997. 1

[6] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software—Practice and Experience*, 2000. 6

[7] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. Draft, March 2001. 1

[8] W. Burton, E. Meijer, P. Sansom, S. Thompson, and P. Wadler. Views: An extension to Haskell pattern matching. `http://www.haskell.org/development/views.html`. 1, 1, 7.1

[9] H. Cirstea, C. Kirchner, and L. Liquori. Matching power. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*. Springer-Verlag, May 2001. 7.2

[10] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. 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. 1

[11] M. de Jonge, E. Visser, and J. Visser. XT: A bundle of program transformation tools. In M. G. J. van den Brand and D. Perigot, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44

of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, April 2001. See also `http://www.program-transformation.org/xt/`. 6

[12] 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. 1

[13] E. Dolstra. First class rules and generic traversals for program transformation languages. Master's thesis, Utrecht University, 2001. 4.2, 6

[14] M. Erwig and S. Peyton Jones. Pattern guards and transformational patterns. In *Haskell Workshop*, 2000. 1, 1, 7.1, 7.1, 7.1

[15] R. Hinze and S. Peyton Jones. Derivable type classes. In G. Hutton, editor, *Haskell Workshop*, Montreal, Canada, September 2000. 1, 7.3

[16] P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 470–482, 1997. 1, 7.3

[17] R. Lämmel. Generic type-preserving traversal strategies. In B. Gramlich and S. Lucas, editors, *Proc. International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*, volume SPUPV 2359, Utrecht, The Netherlands, May 2001. Servicio de Publicaciones - Universidad Politécnica de Valencia. 1

[18] R. Lämmel and J. Visser. Type-safe functional strategies. In *Scottish Functional Programming Workshop*, July 2000. 7.3

[19] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978. 5

[20] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997. 1

[21] S. Peyton Jones, J. Hughes, et al. Report on the programming language Haskell 98, 1999. 1

[22] S. Peyton Jones, N. Ramsey, and F. Reig. C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, October 1999. 8

[23] S. Peyton Jones, A. Reid, T. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *ACM Conference on Programming Languages Design and Implementation*, pages 25–36, 1999. 8

[24] N. Ramsey and S. Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*, June 2000. 8

[25] M. Tullsen. First class patterns. In *2nd International Workshop on Practial Aspects of Declarative Languages*, volume 1753 of *LNCS*, pages 1–15, 2000. 7.1, 7.1

[26] E. Visser. Scoped dynamic rewrite rules. In M. G. J. van den Brand and R. Verma, editors, *Second International Workshop on Rule-Based Programming (RULE'02)*, Firenze, Italy, September 2001. 8

[27] E. Visser, Z. el Abidine Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming (ICFP'98)*, ACM SIGPLAN, pages 13–26, September 1998. 1, 3

[28] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *14th ACM Symp. on Principles of Programming Languages (POPL'87)*, pages 307–313, Munich, Germany, January 1987. 1, 1, 7.1