First Class Rules and Generic Traversals for Program Transformation Languages

Eelco Dolstra
edolstra@students.cs.uu.nl

July 31, 2001

Abstract

The subject of this thesis is the construction of programming languages suitable for the implementation of program transformation systems. First class rules and generic traversals are especially useful in such languages. Stratego, a language specifically intended for program transformations, supports these features, but is untyped and impure.

In this thesis we develop a pure non-strict functional language called RhoStratego, incorporating features from Stratego. First class rules are obtained through the equivalent of Stratego's left-biased choice operator. This approach is not only useful to strategic programming, but is also more powerful than existing proposals to extend pattern matching, such as views and pattern guards. Stratego's generic traversal primitives are implemented through a more fundamental mechanism, the application pattern match, whereby constructed values can be deconstructed in a generic and typeable fashion. We present the syntax and semantics of the language, as well as the semantics of a strict variant.

Furthermore, we have developed a type system for RhoStratego, which consists of the Hindley-Milner type system extended with rank-2 polymorphism and typing rules to support generic traversals. The type system is powerful enough to allow, and ensure the safety of, type unifying and type preserving generic transformations. We have implemented a type checker that infers all types, except rank-2 types for which annotations must be given.

We also discuss the results of the implementation of a compiler for RhoStratego, and in particular how generic traversals and the choice operator can be implemented.

Contents

Chapter 1

Introduction

Motivation The subject of this thesis is the construction of programming languages suitable for the implementation of program transformation systems.

Program transformation is the act of automatically transforming a program in one language into another program in a possibly different language. A recent survey of the field [Vis01b] presents a taxonomy of program transformations, the main types of which are *translations* and *rephrasings*.

In a **translation** a program is transformed to another language. Examples of translations include:

- **Synthesis**, where a program is transformed from a high-level specification to a lower-level specification. The point here is to increase programmer productivity (and program quality) by hiding progressively more details of the machine model, i.e., by going to a higher level of abstraction. The most important form of synthesis is of course **compilation**, where a program is transformed to machine code or some other low-level language. Compilers are a nice example of how program transformation techniques can be used to make the construction of program transformers (in this case, compilers) easier: in the early 1960's, writing a compiler involved many work-years, but due to the emergence of parser generators, code generator generators, etc., nowadays an equivalent compiler can be constructed in a few weeks or months [ASU86]. In **refinement**, a program is derived that implements some domain-specific specification. An example is generating a parser from a grammar.
- **Migration**, where a program is transformed into another language at more-or-less the same level of abstraction; for example, translating Pascal to C.
- **Reverse engineering** is the converse of compilation. Here the goal is to recover (part of) a high-level specification from a lower-level specification. An obvious example is translating machine code back to C.
- **Analysis**, where information about some aspect of a program is extracted, such as control-flow.

In a **rephrasing** a program is transformed into another program in the *same* language. This includes:

- **Normalisation**, where a program is transformed into a program in a sublanguage of the original language. The most common example is **desugaring**, where certain constructs ('syntactic sugar') of the language are eliminated by rewriting them into semantically equivalent fragments of the sublanguage.

- **Optimisation**, which aims to rewrite a program into a more efficient program.
- **Refactoring**, which attempts to improve the design of a program, making it easier to understand and maintain. The converse is **obfuscation**, which makes the program harder to understand.
- **Renovation**, where a certain aspect of a program is improved; an example is (semi-)automatically fixing a Y2K bug.

Program transformation applications can be written in any language; but some make it easier than others. Apart from regular high-level features that make programming easier by allowing specification at a higher level of abstraction (e.g., garbage collection), it turns out that some features are particularly important for program transformation, namely *first class rules* and *generic traversals*. The support for these features in (functional) programming languages is the main subject of this thesis.

Pattern matching gives us the ability to easily deconstruct and inspect values. In a program transformation system programs are typically encoded as tree structures (*terms*) representing the abstract syntax, where the tree nodes are labelled to indicate their meaning (e.g., 'function call', 'variable declaration'); different actions need to be taken based on the shape and kind of the (sub)trees. Pattern matching provides a convenient notation for matching against those trees, and decomposing them into their constituent parts.

We want to be able to write pattern matching code in such a way that we can *separate transformation rules and strategies*. It is a good idea, for reasons of reuse and clarity, to separate the code that actually transforms terms (the *rules*) from the code that specifies how, when, and in what order those rules should be applied (the *strategies*). For example, in an optimiser, we may have a rule expressing commutativity of addition ($x + y = y + x$) or a rule that performs inlining of a function definition. However, such rules, while meaning-preserving, cannot be applied arbitrarily: we may end up with a non-terminating optimiser if we keep applying the commutativity rule, and careless use of inlining may result in a slower and bloated resulting program.

The separation of transformation rules and strategies requires that the rules are *first class*. For example, in Haskell we can write transformation rules using pattern matching λ-abstractions, i.e., $\lambda pat \rightarrow expr$. However, if the pattern fails to match, the program *diverges* (fails entirely). Hence, pattern matching λ-abstractions are not first class in Haskell. Alternatively, we could use case-expressions, but these bind different transformation rules together, and so are not first class either.

Generic traversals are the other major feature. Transformations often need to be applied at many points in the abstract syntax tree. We should have generic operations that can traverse arbitrary data structures, applying arbitrary transformations in some predefined way (such as bottom-up or top-down).

In this thesis, we first take a look at two languages that are particularly useful for program transformation: Stratego and Haskell. Because these languages both have certain pros and cons with regard to this subject, we then develop a new pure, non-strict, strongly typed functional language called RhoStratego that attempts to combine their advantages.

Outline This thesis is structured as follows. In chapter 2 we look at Stratego, a language that is specifically intended for program transformation applications, and discuss its advantages and disadvantages. In chapter 3 we turn to the question of whether those disadvantages can be overcome while maintaining Stratego's advantages in the pure functional language Haskell (chapter 3). Since this is not the case,

we present in chapter 4 a new functional language called RhoStratego, and give its syntax and semantics. We also compare RhoStratego's pattern matching facilities to those found in other languages. In chapter 5 we discuss compilation issues for RhoStratego and describe the RhoStratego compiler. In chapter 6 we develop a type system for RhoStratego. A somewhat larger example of a RhoStratego program is given in chapter 7. The annotated source code of the RhoStratego interpreter, type checker, and standard library appears in the appendices.

Acknowledgements I would like to thank my supervisor Eelco Visser for the many interesting conversations we had during the course of this project, and for reading and commenting upon this thesis. I would also like to thank Arthur Baars for pointing out that pattern match failure in the `do`-notation will result in the monad's `fail`-method being called (chapter 3). Finally, I am grateful to Paul Harrenstein who provided the LaTeX environment for typesetting natural deduction proofs.

Chapter 2

Program transformation in Stratego

Since programs are naturally encoded as *terms*, *term rewriting* [BN98] provides a natural means of expressing transformations on programs. A term rewriting system (TRS) consists of a number of *rewrite rules* $l \rightarrow r$, where l and r are terms containing variables. A *term* is a function symbol applied to zero or more terms. For example, the following two rewrite rules express simple transformations over numerical expressions:

$$\begin{array}{rcl} \texttt{Plus}(x, y) & \rightarrow & \texttt{Plus}(y, x) \\ \texttt{Plus}(x, \texttt{Zero}) & \rightarrow & x \end{array}$$

We can *apply* a rewrite rule $l \rightarrow r$ to a term t by *matching* t against the pattern l. If the match succeeds, we replace t by r, applying any substitutions found in the match. A rewrite rule can also be applied to a subterm of a term. A term t_1 *reduces* to t_2 if there is a sequence of applications of rewrite rules that transforms t_1 to t_2. A term is in *normal form* when no rules apply. For example, the term `Plus(Zero, Succ(Zero))` can be normalised to `Succ(Zero)` by applying the first and then the second rule.

A TRS is *confluent* when no term has more than one normal form, it is *terminating* if there are no infinite sequences of rewrite rules, and it is *convergent* if it is both confluent and convergent, i.e., if every term has a unique normal form. Clearly, the TRS given above is not terminating, since the first rule is always applicable to a `Plus` term.

Since most interesting rewrite systems (e.g., optimisers) are not confluent and/or terminating, simply applying rules randomly is insufficient. We need to *control* the rewriting process by specifying how and when the rewrite rules are to be applied; we need to specify a *rewriting strategy*. Since no particular strategy is suitable for all purposes, we need a rewriting language that allows the specification of rewrite rules along with the strategies controlling how the rewrite rules are applied. This is precisely the concept underlying Stratego [VeABT98]. It also supports generic traversals: the ability to apply transformations in arbitrary terms according to some traversal strategy. In the remainder of this chapter we give an overview of Stratego. Complete information can be found in [Vis, Vis01a].

2.1 Terms and rewriting

Terms Unsurprisingly for a system based on term rewriting, Stratego programs operate on *terms*. A Stratego term is either an integer or string constant, or a *constructor* (or *function symbol*, as they are called in rewriting literature) applied to zero or more terms (called *subterms*).

Example 1 (Terms) The following are terms:

```
Plus(Succ(Zero), Succ(Succ(Zero)))
Def("id", Lambda("x", Var("x")))
```

Strategies A Stratego program (or *specification*) consists of a set of named *strategies*. A strategy rewrites an unnamed term called the *current term*. Stratego provides the following primitive strategies:

- The **match** primitive, written as `?`*pat*, matches the current term against a *pattern*. A pattern is a term that may contain variables. If a match is successful, the current term is unaffected and the variables (if any) become bound in the current scope (see below) to the matching (sub)terms of the current term. If a match fails, the current term is replaced by the special symbol δ ('fail').
- The **build** primitive, written as `!`*term*, replaces the current term with the specified term.

Example 2 (Match and build) The following strategy will match successfully against `Plus(Zero, Succ(Succ(Zero)))` (the number 2 in Peano-style), with `x` becoming bound to `Succ(Succ(Zero))`:

```
s = ?Plus(Zero, x)
```

On the other hand, the same match applied to the term `Plus(Succ(Zero), Succ(Succ(Zero)))` will yield δ.

The following strategy builds a term:

```
s = !Plus(Succ(Zero), Succ(Succ(Zero)))
```

Combining strategies Strategies can be combined using the following primitive strategy operators:

- **Sequential composition**, written as s_1 `;` s_2, applies s_1 to the current term, and if it is successful (i.e., does not yield δ), then s_2 is applied to the result of s_1. If s_1 yields δ, then the result of the composition is δ. In pseudo-notation, making the current term explicit and viewing strategies as functions from terms to terms, its semantics is as follows:

$$(s_1 \texttt{ ; } s_2)\ t = \begin{cases} s_2\ (s_1\ t) & \text{if } s_1\ t \neq \delta \\ \delta & \text{otherwise} \end{cases}$$

- **Left choice** (also referred to as *left-biased choice*), written as s_1 `<+` s_2, applies s_1 to the current term, and if that is *unsuccessful*, applies s_2 to the (original) current term. In pseudo-notation:

$$(s_1 \texttt{ <+ } s_2)\ t = \begin{cases} s_1\ t & \text{if } s_1\ t \neq \delta \\ s_2\ t & \text{otherwise} \end{cases}$$

- **Indeterministic choice**, written as s_1 `+` s_2, is like left choice, except that it *may* reverse the arguments, i.e., try s_2 first and do s_1 if s_2 fails. The order in which the arguments are tried is left undefined[1]. The advantage of indeterministic choice is that it gives the compiler the freedom to rearrange matches to achieve greater efficiency. In pseudo-notation:

$$(s_1 \texttt{ + } s_2)\ t = \begin{cases} s_1\ t & \text{if } s_1\ t \neq \delta \\ s_2\ t & \text{if } s_2\ t \neq \delta \\ \delta & \text{otherwise} \end{cases}$$

  Note that the conditions in the right-hand side of the first two clauses are not mutually exclusive.

Example 3 (Peano axioms) The Peano axioms for addition can be expressed as follows:

```
plus0 = ?Plus(Zero, x); !x
plusN = ?Plus(Succ(x), y); !Plus(x, Succ(y))
```

Rules *Rules* are strategies written in a notation that is more conventional in term rewriting. A rule has the form $pat \rightarrow term$ where the current term is matched against the pattern and replaced by the right-hand side, if the match is successful. It is syntactic sugar for the sequential composition `?`pat `; !`$term$.

Example 4 (Rules) The previous example can also be written as

```
plus0: Plus(Zero, x) -> x
plusN: Plus(Succ(x), y) -> Plus(x, Succ(y))
```

As said before, strategies are named, and they can therefore invoke each other. In addition, strategies are *higher-order*: they can be passed to other strategies as *strategy parameters*.

Example 5 The standard strategy operator `try` applies a strategy `s` to the current term, leaving the current term unmodified if `s` fails:

```
try(s) = s <+ id
```

where `id` is the identity strategy.

It should be noted that while strategies can be passed as arguments to other strategies, terms can only be passed to strategies as the current term or as a build strategy (e.g., `f(!t)`).

Strategy application and where-clauses While strategies are usually applied to the current term, they can also be applied to subterms. Stratego also has *where-clauses* that allow terms to be computed and named separately from the current term.

Example 6 The following strategy:

```
s = where (<plusN> Plus(Succ(Zero), Zero) => x);
    !Plus(<plus0> Plus(Zero, Zero), x)
```

will result in the term `Plus(Zero, Plus(Zero, Succ(Zero)))`.

[1] In the current implementation of Stratego the left argument is tried first, so indeterministic choice is identical to left choice. This behaviour should not be relied on, however, as it is implementation dependent.

Recursion Stratego currently only support *explicit* recursion. A strategy s is allowed to call itself neither directly nor indirectly. Rather, recursion must be specified through the *recursion operator* `rec`.

Example 7 (Recursion) The function `map` that applies a strategy to all elements of a list can be expressed as follows:

```
map(s) = rec x(\[] -> []\ + \[y | ys] -> [<s> y | <x> ys]\)
```

`[]` and `[x|xs]` are the constructors of a regular nil/cons-list. A rule between backslashes (e.g., `\[] -> []\`) is an *anonymous* rule.
The standard strategy `repeat` applies a strategy until it no longer succeeds.

```
repeat(s) = rec x(s; x <+ id)
```

Note that sequential composition binds stronger than the choice operators. We can use `repeat` to do Peano-style addition:

```
eval = repeat(plus0 + plusN)
```

Scope By default, the match operator will bind variables throughout the entire strategy. This means that for example `map(?Foo(x); !Bar(x))` will probably not do what is expected, i.e., transform all `Foo`s into `Bar`s; all the `x`s in the list have to be equal as well. We can write `map({x: ?Foo(x); !Bar(x)})` to restrict the scope of `x` to the strategy argument of `map`. Alternatively, we can use an anonymous rule `map(\Foo(x) -> Bar(x)\)`; the variables occurring in the left-hand side of the rule are only in scope in the rule itself.

Congruences Congruences are syntactic sugar for a common transformational pattern. A congruence `C(`s_1`, ..., `s_n`)` is equivalent to the rule `C(`x_1`, ..., `x_n`)` → `C(<`s_1`> `x_1`, ..., <`s_n`> `x_n`)`; that is, we specify the n strategies to be applied to the subterms of an n-ary term.

Example 8 The previous example can be written more succinctly as:

```
map(s) = rec x([] + [s | x]);
```

The following strategy applies a strategy `s` to all the values in the leaves of a binary tree:

```
t(s) = rec x(BinTreeNode(x, x) + BinTreeLeaf(s))
```

Note that there is no collision between `C(...)` used as a congruence, a build, or a match, since the latter two must be prefixed by a `!` or `?` operator, respectively.

2.2 Generic traversals

Generic traversals are constructed from two primitive strategy operators: `all` and `one`. `all` applies a strategy to all children of a term; it fails if the strategy fails to apply to at least one child. `one` applies a strategy to exactly one child of a term; it fails if there is no term to which the strategy applies.

Example 9 `all` and `one` allow more complex traversals to be constructed.

```
topdown(s)  = rec x(s; all(x))
bottomup(s) = rec x(all(x); s)
oncetd(s)   = rec x(s <+ one(x))
oncebu(s)   = rec x(one(x) <+ s)
```

`topdown` and `bottomup` apply a strategy to all subterms in a top-down or bottom-up fashion, respectively; `oncetd` and `oncebu` apply a strategy to exactly one subterm. Simple examples of the use of these strategy operators include:

```
rename = topdown(try(Var("x") -> Var("y")))
simplify = topdown(repeat(plus0))
occurs_x = oncetd(?Var("x"))
```

The latter strategy fails if a variable named `x` does not occur in the current term.

Generic algorithms Using generic traversals it is possible to specify generic algorithms. Some classes of transformations appear very often, no matter what the language being transformed is. Examples are renaming, performing substitutions, and inlining. Rather than write specific code for each language, it is possible to write these transformations generically; then they only have to be parameterised with the specifics of each language. Generic implementations of renaming and substitution are described in [Vis00] (these are used in section B.2).

2.3 XT

Stratego is part of XT, the Program Transformation Tools [dJVV01]. XT bundles a large number of tools for program transformation, including:

- The **ATerm** library [vdBdJKO00] is a C library for working efficiently with tree-structured terms, and is used by Stratego. It can read and write terms in textual and binary formats. The ATerm file format is the glue that holds the XT tools together. An interesting aspect of the ATerm library is its *maximal sharing*: if two terms are equal, then they have the same memory address. This often reduces memory usage substantially and may increase performance (because equality tests are very cheap).
- **sglr** is a Scannerless Generalised LR parser. The fact that it is scannerless means that the lexical syntax is specified in the same language as the context-free syntax; there is no need to use a separate lexical analyser generator.
- **Grammar Base** is a large collection of ready-to-use grammars for `sglr`. It includes grammars for C, Cobol, Haskell, Java, XML, and many others.

2.4 Conclusion

Stratego has the following advantages:

- Genericity. Stratego makes it very easy to traverse a term and transform it.
- It is untyped. This is sometimes nice because:
  - It allows rapid development of transformations. Sometimes, for example, we want to write transformations that apply to only a small part of the abstract syntax (say, in a desugarer). Not having to specify data types for the entire abstract syntax saves time.

  - It allows terms of different types to be mixed freely.

- Congruences make it very it easy to express certain kinds of transformations on terms (namely, those that are sort-preserving).
- It is smoothly integrated (through the ATerm interface) into the XT toolbox.

On the other hand, it has a number of disadvantages:

- It is untyped. A substantial amount of development time is lost finding trivial bugs that could have been detected by a type checker. Type checking gives a proof of a certain kind of correctness of a program; the lack of a type discipline means that there are very few static guarantees with regard to correctness. Unfortunately, typing Stratego is not very easy, but there is progress in this area [Läm01].
- The language has two kinds of objects, namely, terms and strategies; the strategies operate on the terms. These are strictly separated; and so it is not possible, for example, to pass a strategy to another strategy as part of a term; strategy arguments are required to obtain higher-order strategies.

These two disadvantages are solved in modern functional languages. After all, such languages are strongly typed and are founded upon the λ-calculus [Bar84]. The main feature of the λ-calculus is precisely that it unifies terms and the things operating on terms (i.e., data and code). Functional languages also typically have extensive pattern matching facilities. In the next chapter we look at whether strategic programming is easily possible in one particular functional language.

Chapter 3

Strategic programming in Haskell

Functional programming languages are popular for program transformation applications for several reasons:

- They typically have extensive pattern matching facilities.
- They usually have type systems that make it easy to declare complex data types.
- Recursion and a 'cheap' function syntax make it easy to write code to traverse through data values.

In the previous chapter we looked at the features that Stratego offers to support program transformation. In this chapter we turn to the question of whether such features can be implemented elegantly in a typical modern functional language, namely Haskell.

Haskell 98 [PH+99] is a pure non-strict functional language. It has algebraic data types, an extended Hindley-Milner type system with parametric polymorphism and type classes, along with many other features. In particular, it has pattern matching facilities.

3.1 Rewriting

It is not difficult to implement Stratego-like functionality in Haskell. It would be nice if we could write a Stratego rewrite rule such as:

```
plus0: Plus(Zero, x) -> x
```

as

```
plus0 = \(Plus Zero x) -> x
```

assuming a data type

```
data Exp = Zero | Succ Exp | Plus Exp Exp
```

Of course, this will not work because we cannot handle pattern match failure in λ-abstractions: if a pattern fails to match, the program is said to *diverge*; in other

words, the program crashes. This fact points out that pattern matches are not 'first class' in Haskell. The solution is to make failure explicit by lifting the result into a `Maybe` type[1]:

```
plus0 (Plus Zero x) = Just x
plus0 _ = Nothing
```

In general, then, a strategy has the following type:

```
type St a b = a -> Maybe b
```

Strategy operators Stratego's left choice and composition operators (here written as `<*` and `<+`, respectively) are easy to define:

```
(<+) :: St a b -> St a b -> St a b
(s1 <+ s2) t = maybe (s2 t) Just (s1 t)

(<*) :: St a b -> St b c -> St a c
s1 <* s2 = \t -> maybe Nothing s2 (s1 t)
```

Indeterministic choice can be accomplished by returning a list of results, but we will not do so here.

Combining strategies is just as easy as in Stratego:

```
idS :: St a a
idS t = Just t // identity strategy

tryS :: St a a -> St a a
tryS s = s <+ idS

repeatS :: St a a -> St a a
repeatS s = tryS (s <* repeatS s)
```

Writing the basic rewrite rules, however, is tiresome because we spend a lot of time packing and unpacking the `Maybe` wrapper. For example, the congruence over a constructor `Plus Exp Exp` would look like this:

```
cgrPlus :: St Exp Exp -> St Exp Exp -> St Exp Exp
cgrPlus s1 s2 (Plus e1 e2) = maybe Nothing (\e1' ->
  maybe Nothing (\e2' -> Just (Plus e1' e2')) (s2 e2)) (s1 e1)
cgrPlus s1 s2 _ = Nothing
```

Fortunately, Haskell's *do-notation* provides us with a much nicer way to write the above. The `Maybe` type forms a monad [Wad92], with the monadic *bind*, *unit* ('return') and *fail* operators defined as follows:

```
Just x  >>= k = k x
Nothing >>= k = Nothing
return        = Just
fail s        = Nothing
```

Therefore we can write `cgrPlus` as:

[1] The data type `Maybe` is defined as `data Maybe a = Nothing | Just a` where `Nothing` denotes failure and `Just a` denotes success with a value `a`.

```
cgrPlus s1 s2 t
  = return t >>= \t' ->
    case t' of
      Plus e1 e2 ->
        s1 e1 >>= \e1' ->
        s2 e2 >>= \e2' ->
        return (Plus e1' e2')
      _ -> fail ""
```

Using the do-notation gives us the following, equivalent code:

```
cgrPlus s1 s2 t
  = do Plus e1 e2 <- return t
       e1' <- s1 e1
       e2' <- s2 e2
       return (Plus e1' e2')
```

This is quite a bit more readable than the original. This code works by virtue of the fact that pattern match failure in the left-hand side of do-bindings (`Plus e1 e2 <- ...`) will not lead to global divergence but will instead invoke the `fail` method of the `Maybe` type. This is because the `pat <- ...` construct in the do-notation is sugar for `case ... of pat -> ...; _ -> fail`.

3.2 Generic traversals

Another issue is how to implement generic traversals. Since Haskell lacks generic features, we have to describe explicitly for each data type how to traverse it. This can be done using type classes. In Stratego, the strategy `all(s)` applies `s` to *all* subterms of the current term. This poses a problem in Haskell since not all subterms will have the same type. We will therefore restrict `all` to operate only on subterms of the same term as the root.

```
class Trav a where
  allS :: St a a -> St a a
```

Such functions as `bottomup` can then be defined in the expected way:

```
bottomupS :: Trav a => St a a -> St a a
bottomupS s = allS (bottomupS s) <* s
```

An instance might look like this:

```
instance Trav Exp where
  allS s t@Zero = return Zero
  allS s t@(Succ _) = cgrSucc s t
  allS s t@(Plus _ _) = cgrPlus s s t
```

Example 10 (Evaluation) The following code evaluates an expression by rewriting it bottom-up:

```
plus0 t = do Plus Zero x <- return t
             return x

plusN t = do Plus (Succ x) y <- return t
```

```
                return (Plus x (Succ y))

eval = bottomupS (repeatS (plus0 <+ plusN))
```

There are more refined (and complex) approaches [LV00], but these also suffer from the fact that a lot of code must be written for each data type over which we want to traverse.

3.3 Derivable type classes

Generic or polytypic programming makes it possible to write functions that operate on different data types. Derivable type classes [HP00] make it possible to write functions such as `show` or `(==)` once for all data types. In standard Haskell, such functions must be written explicitly for all data types for which we want to support them. For this reason Haskell provides the *ad-hoc* `deriving` mechanism, which only works for a number of standard classes (such as `Show` or `Eq`). Derivable type classes allow the programmer to provide a *default* method that operates on a universal data type. The default method is used for all types for which no specific method is given.

For example, consider the following abstract syntax:

```
data Exp = Var Id | Plus Exp Exp | Min Exp Exp
data Id = Id String
```

Suppose that we want to traverse a term and replace every identifier `x` by `y`. In Stratego we would write:

```
rename = topdown (try (Id("x") -> Id("y")))
```

and this works for all data types. Using derivable type classes we can accomplish the same:

```
class Rename a where
  rename :: a -> a

  rename {| Unit |}    Unit      = Unit
  rename {| a :+: b |} (Inl x)   = Inl $ rename x
  rename {| a :+: b |} (Inr y)   = Inr $ rename y
  rename {| a :*: b |} (x :*: y) = (rename x) :*: (rename y)

instance Rename Exp

instance Rename Id where
  rename = \(Id x) -> Id (if x == "x" then "y" else x)
```

The default definition for `rename` works as follows. We can view each data type as a combination of sums and products of other types. For example, `Exp` can be seen as `(Id + Exp * Exp) + Exp * Exp`. By providing clauses for units (types that are not instances of the class), the left and right alternatives of a sum, and products, we completely cover any data type.

Hence, the generic definition of `rename` traverses the term, applying `rename` to all children that are instances of the class `Rename`. To actually accomplish the desired result of replacing `x` by `y`, we *override* the default definition of `rename` with a specific definition for the type `Id`.

The problem with this approach is that the class `Rename` is tied to a particular transformation, namely renaming `Id`s. If we want to do a different type of generic transformation, we have to define a new class, providing different instances for the appropriate types. We cannot write a general class `Topdown` containing a method that is parameterised 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 down the function `id`. Therefore derivable type classes do not make it easy to write many of the kinds of generic traversals that occur in program transformation systems.

3.4 Conclusion

In this chapter we have seen that while Haskell gives us some of the things we need in a language for program transformation, it is not enough. In particular, some of Stratego's features — first class pattern matching and generic traversals — are missing. In the next chapter we develop a functional language that does have these things.

Chapter 4

The RhoStratego programming language

All language designers are arrogant. Goes with the territory...
— Larry Wall

In the previous two chapters we have looked at two useful languages for program transformation: Stratego and Haskell. In this chapter we present the new language RhoStratego which combines the purity and higher-order features of functional programming with the first class pattern matching and generic traversals offered by Stratego.

We first give an overview of RhoStratego. Next, we present the syntax and semantics of the language (including the semantics of a strict variant of the language). Finally, we contrast RhoStratego's pattern matching facilities to those in other functional languages, and compare RhoStratego to the ρ-calculus.

4.1 Overview

4.1.1 The basics

RhoStratego is a non-strict purely functional programming language. It consists essentially of the λ-calculus extended with constructors, pattern matching, and let-bindings.

Example 11 (Simple values) A RhoStratego program is a list of *definitions*. A definition binds a name to a value. Here are some straight-forward definitions:

```
id = x -> x;
const = x -> y -> x;
x = id 123; // evaluates to 123
. = f -> g -> x -> f (g x);
```

Note that there is no symbol that syntactically starts the λ-abstraction (such as a λ or Haskell's backslash). This follows the notation used in term rewriting.

Example 12 (Constructors) The language has constructors that can be deconstructed using a pattern match:

```
f = (Foo Bar x -> x) (Foo Bar Fnord); // evaluates to Fnord
```

Following Haskell's tradition, constructor names start with capitals.

4.1.2 Choice

RhoStratego inherits Stratego's left choice operator (`<+`). The intended semantics is that this operator evaluates its left argument, and if it 'fails', returns its right argument. An expression is said to fail when a pattern match failure occurs somewhere in the evaluation of the expression.

Example 13 (Left choice) The choice operator allows us to distinguish between different constructors, and to write first class transformation rules. The Peano rules from chapter 2 can be expressed as follows:

```
plus0 = Plus Zero x -> x;
plusN = Plus (Succ x) y -> Plus x (Succ y);
onestep = plus0 <+ plusN;
```

Note that the choice operator binds weaker than λ-abstraction. This allows us to omit parentheses in the most common cases.

One may not be convinced that the `onestep` function in the previous example works, and in fact it does not without the addition of some semantic machinery. After all, if we apply `onestep` to, say, `Plus (Succ Zero) Zero`, the choice operator will evaluate its left argument, namely `Plus Zero x -> x`, which is immediately in normal form and hence does not fail. Therefore `(Plus Zero x -> x <+ ...) (Plus (Succ Zero) Zero)` will evaluate to `(Plus Zero x -> x) (Plus (Succ Zero) Zero)`, which will yield a pattern match failure.

One solution is to write `onestep` as follows:

```
onestep = x -> (plus0 x <+ plusN x);
```

That is, it pushes the argument into the choice alternatives to ensure that the pattern match (and the pattern match failure) occurs at the right time, namely in the scope of the choice operator.

While this approach works, it is not quite satisfactory. After all, the choice operator in Stratego is used to choose between strategies, which are functions from terms to terms; so we do not want to choose between functions but between functions *applied to terms*.

The solution, then, is rather simple: automate the transformation from the first definition of `onestep` to the second. We do this by adding the following rule, which *distributes* the argument to a choice over the choice alternatives:

$$\textsc{Distrib} : (e_1 \texttt{ <+ } e_2)\ e_3 \mapsto e_1\ e_3 \texttt{ <+ } e_2\ e_3$$

It is easy to see that this does in fact transform the first `onestep` into the second (after η-expanding the body of `onestep`).

Example 14 (More choices) Here are definitions of some popular functions:

```
foldr = op -> nul ->
  (Nil -> nul <+ Cons x xs -> op x (foldr op nul xs));

map = f -> foldr (x -> xs -> Cons (f x) xs) Nil;

if = True  -> e1 -> e2 -> e1 <+
     False -> e1 -> e2 -> e2;
```

Cuts Of course, adding the DISTRIB rule just gives us the opposite problem: what if we actually want to choose between functions? For example, suppose that we write `if` like this:

```
if = True -> e1 -> e2 -> e1 <+
     _    -> e1 -> e2 -> e2;
```

which seems reasonable, since if a boolean value is not true then it must be false. Now suppose we wrote the following code to choose between two functions:

```
x = (if (condition) (A -> B) (C -> D)) C;
```

Presumably, if the condition evaluates to `True`, then `A` $\rightarrow$ `B` will be selected, which will fail to match against `C`, and `x` will evaluate to `fail`. What *really* happens, however, is that since `A` $\rightarrow$ `B` is a function, `C` (as well as the direct arguments to `if`) will be pushed into the choice, and we get:

```
x = (True -> e1 -> e2 -> e1) True (A -> B) (C -> D) C <+
    (_    -> e1 -> e2 -> e2) True (A -> B) (C -> D) C;
```

This subsequently reduces to:

```
x = (A -> B) C <+
    (_ -> e1 -> e2 -> e2) True (A -> B) (C -> D) C;
```

and:

```
x = fail <+
    (_ -> e1 -> e2 -> e2) True (A -> B) (C -> D) C;
```

so we incorrectly end up choosing the `False` clause:

```
x = (_ -> e1 -> e2 -> e2) True (A -> B) (C -> D) C;
```

which ultimately evaluates to `x = D`.

A related problem is the *scope* of the choice operator over its left argument. For example, suppose we want to distinguish a term between two alternatives, say the constructor `C`, and everything else, and do something different depending on that. We could write something like `(C -> e1) <+ (x -> e2)`. However, if `e1` fails, RhoStratego will backtrack to `x -> e2`; this may be what we want, but very often it is not, and we just want to choose between some alternatives and not backtrack. In other words, the scope of the backtracking should, in this case, be limited to the pattern match, and failure in `e1` should not be caught (at this level; we may want to deal with it in some outer choice). This problem also manifests itself in Stratego; it concerns the *scope* of the choice operator over its left argument. We want a way to specify, for example, that failure in the pattern match `C` should be caught, but not failure in the body of the function.

The connection between the above two problems is that in both cases we want to return a value 'as is', without ever going to the right-hand side choice alternative; but the choice operator happens to treat functions and `fail` values specially. Both of the above problems can be solved by wrapping the values that we want to leave untouched by the choice operator inside a dummy constructor. The first example, for example, would become:

```
x = (S x -> x) ((if (condition) (S (A -> B)) (S (C -> D))) C);
```

Since the choice operator does not touch the value `S (A → B)`, it escapes out of the choice, and is finally unpacked by `S x → x`. The second example becomes

```
f = y -> (S x -> x) ((C -> S e1 <+ x -> S e2) y);
```

This wrapping and unwrapping is rather inconvenient. Furthermore, in the spirit of the separation of rewrite rules and strategies, one should be able to write the second example as

```
a = C -> e1;
b = x -> e2;
f = a <+ b;
```

so that `a` and `b` can be used independently. But when we wrap `e1` and `e2` into a dummy constructor, every calling site must be adapted to handle this. This hampers reuse.

RhoStratego solves these problems through the *cut* mechanism[1]. The unary operator `^` 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 examples above can be written as:

```
x = (if (condition) ^(A -> B) (C -> D)) C;
f = C -> ^e1 <+ x -> e2;
```

Note that the cut only needs to be applied to the left-hand side.

For the exact semantics of a cut there are two alternatives. The first is that a cut is just sugar for the constructor wrapper hack described above, except that the choice operator is modified to remove cuts automatically. In that case, however, we cannot write something like `^1 + ^2`; that is, we still have the problem that rules and strategies cannot be separated cleanly. Furthermore, if `^e` is distinguishable from `e`, then this fact should be reflected in the type system.

The second alternative is that cuts are *transparent*; in any context except the left-hand side of a cut they 'disappear' automatically, meaning that `^1 + ^2` will evaluate to `3`, and the pattern match `(A → B) ^A` will succeed. This is the approach taken in RhoStratego.

Failure The constant `fail` can be used to created failure outside of a pattern match. Furthermore, `fail` can occur in pattern matches. A function with such a pattern will return the body of the function if the argument evaluates to `fail`; otherwise `fail` will be returned.

Example 15 (Strict application) The function `st` applies a function to an argument, evaluating the argument first:

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

Note that we must write `^fail` in order to prevent `fail` from being 'caught' by the choice operator.

Example 16 (Strategy operators) Several of Stratego's non-generic strategy operators can now be defined:

```
| = f -> g -> t -> st g (f t);
try = s -> (s <+ id);
repeat = s -> try (s | repeat s);
```

[1]The name comes from Prolog, where cuts are used to limit backtracking

The first function, `|` ('pipe'), performs strict sequential composition; this is Stratego's `;` strategy operator. We cannot use regular sequential composition in `repeat`, i.e.,

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

as this will cause `repeat` to get stuck in an infinite recursion: `repeat s` expands into `try (s | repeat s)`, which is `(s | repeat s) <+ id`. Since `s | repeat s` is equal to `t -> (repeat s) (s t)`, and since 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.

Using `repeat`, we can of course write:

```
eval = repeat (plus0 <+ plusN);
```

4.1.3 Generic traversals

Stratego accomplishes generic traversals over terms by means of the `all` and `one` traversal primitives. RhoStratego does not have these primitives. Instead, generic traversals are obtained through *application pattern matches*. This is a pattern of the form `c x`, where `c` and `x` are both variables. This will match with a constructor application, and `x` will be bound with the last argument of the constructor (the *suffix*) and `c` will be bound to the constructor applied to the other arguments (the *prefix*).

Example 17 (Application match) The following definitions show the use of application match patterns:

```
f = (c x -> c) (Foo Bar Fnord); // evaluates to Foo Bar
g = (c x -> x) (Foo Bar Fnord); // evaluates to Fnord
```

The application match pattern allows us to look at the immediate subterms of a term in a linear fashion, just like traversing a nil/cons-list: the `c x` pattern corresponds to matching a cons. Then what corresponds to matching with nil? The answer is that that is simply the pattern `x`: if the match `c x` fails, then the argument is either a constructor (without arguments) or another normal form (such as a function or an integer literal).

Example 18 (Getting the immediate subterms) The following function returns the subterms of the argument as a list:

```
kids = c x -> Cons x (terms c) <+ x -> Nil;
```

Note that this function is untypeable in the type system given in chapter 6.

Example 19 (Term size) This function determines the size of a term (i.e., the number of constructor nodes and base values):

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

With this single traversal primitive, we can formulate Stratego's `one` and `all` primitives.

Example 20 (`all` strategy operator) Stratego's `all` operator, which applies a function `f` to all immediate subterms of some term, can be implemented as follows:

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

Here is how it works. We first check whether the argument is a constructor application. If it is not, and the argument is either a constructor or some other value (e.g., an integer), we do nothing (i.e., identity, `c` → `c`). If it is, we recursively apply `f` to the subterms in the left-hand side of the constructor application (in `all f c`). The result of this will be `C (f` e_1`) .... (f` e_{n-1}`)`, where n is the number of arguments to the constructor. Applying this expression to `f x` will complete the construction of the new term `C (f` e_1`) .... (f` e_n`)`. Since the semantics of `all` in Stratego is that the strategy `f` must be *successfully* applied to all subterms, we decree that this must be a *strict* application. Finally, the result must be cut in order to prevent failure in `f` from warping us into `c` → `c`.

Example 21 (`one` strategy operator) Stratego's `one` operator, which applies a function `f` to exactly one immediate subterm of some term and fails if `f` cannot be applied to at least one such term, can be implemented as follows:

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

The idea is that we deconstruct the value, going from right to left, until we find a subterm to which `f` can be successfully applied. If that is the case, then we apply the unmodified left-hand side `c` to `f x`. We do this by means of the strict application `st c (f x)`. If `f x` fails (i.e., if the strict application fails), we must look in `c` for some other subterm to which `f` can be applied (in `one f c`). If we reach the constructor (that is, if the application pattern match `c x` fails), then there is no subterm to which `f` applies, and `one` fails.

As an example, consider `one (A → B) (C A A B)`. Since the term `C A A B` can be deconstructed into `c = C A A` and `x = B`, we end up in the left-hand side of the outer choice, and we try `f x` ≡ `(A → B) B`. This fails, so we recursively evaluate `one (A → B) (C A A)`. Here the argument can also be deconstructed (into `c = C A` and `x = A`), but now `f x` ≡ `(A → B) A`, which is successful and yields `B`. Therefore, the result of `one (A → B) (C A A)` is `C A B`, and the result of `one (A → B) (C A A B)` is `C A B B`.

Now consider `one (A → B) (C B)`. Applying `A → B` to `B` fails, so we recurse into `one (A → B) C`. `C` is not an application, however, and we end up in the clause `c → fail`. Consequently, `one (A → B) (C B)` ≡ `fail`.

Example 22 (Hyperstrictness) The following function enforces hyperstrictness of its argument, i.e., evaluates it entirely:

```
force = all force;
```

This works because `all` evaluates the constructor application spine. Note that even leaf terms like integer literals are evaluated since they too are matched against the `c x` pattern.

Example 23 (Generic strategy operators) Since we can write `all` and `one`, we can also write the Stratego strategy operators that are expressed in terms of `all` and `one`. For example:

```
topdown = s -> s | all (topdown s);
bottomup = s -> all (bottomup s) | s;
oncetd = s -> (s <+ one (oncetd s));
```

Of course, we can use these functions. For example, the following functions succeeds only if the variable `x` occurs in a term:

```
occurs = x -> oncetd (Var y -> if x == y then Var y else fail);
```

4.1.4 Syntactic sugar

RhoStratego provides syntactic sugar for congruences, lists, and tuples.

Example 24 (Congruences) Congruences are a very useful feature in Stratego, and RhoStratego has them as well. The following example applies a function to all the values in the leaves of a binary tree.

```
f = g -> (@BinTreeNode(f g, f g) <+ @BinTreeLeaf(g));
```

This is syntactic sugar for

```
f = g -> (BinTreeNode l r -> BinTreeNode (f g l) (f g r)
       <+ BinTreeLeaf x -> g x);
```

Example 25 (Lists and tuples) RhoStratego has sugar for lists and tuples.

```
numbers = [1, 2, 3];
stuff = <42, "Foo", C 23>;
```

Note that the use of angle brackets, instead of parentheses, allows the unambiguous notation of unary tuples.

4.1.5 Input and output

RhoStratego has a monadic I/O system [PW93]. An I/O action is a function of type `IO a`, which is just a synonym for `World -> <World, a>` (World is just a dummy value representing the state of the world; it ensures proper sequencing of I/O actions). The operator `(>>=)` ('bind') is used to sequence two I/O actions. Note that RhoStratego as yet has no concept of data hiding, so it is possible to 'cheat' (i.e., perform unsafe I/O).

A RhoStratego program is executed by performing the I/O action returned by a function called `main` of type `IO <>`.

Example 26 (Hello World) This complete RhoStratego program print 'Hello World!' on standard output.

```
main = (putStr "Hello ") >> (putStr "World!");
```

Note that `a >> b = a >>= x -> b`.

The runtime system provides functions to read and write ATerms. This is essential for integration into XT. Function symbols appearing in the ATerm are mapped at runtime to constructors. For example, the ATerm `Foo(Bar("x"), 123)` is mapped to `Foo (Bar "x") 123`. Constructors must be declared using the syntax given in chapter 6; otherwise they will not be known to the runtime system.

Example 27 (ATerms) This program reads an ATerm, modifies it slightly, and writes it out as a textual ATerm.

```
main =
  let a1 = readTerm "in.trm";
      a2 = Foo x -> writeTerm "out.trm" ATFmtText (Bar x);
  in a1 >>= a2;
```

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

Figure 4.1: RhoStratego abstract syntax

4.1.6 Module system

The 'module system' (such as it is) is identical to that of C/C++. Source files can be compiled separately into object files; the object files are then linked together to form the complete program. One source file can refer to functions defined in another source file by declaring its type signature (see chapter 6) but not its definition. Modularity is achieved through a simple textual file inclusion facility provided by a *preprocessor* that is applied to the source file before it is fed into the compiler; public function definitions are placed in *header files* that can be *included* by other files. Inlining can be accomplished by putting a function's definition in a header file.

Example 28 (Standard library) RhoStratego provides a standard library which provides a number of useful functions. The standard library is compiled separately; user programs can use its functionality by including the file `stdlib.rh`. For example, the file `stdlib.rh` contains the following type signature (among others):

```
id :: a . a -> a; // i.e., id is polymorphic
```

The file `stdlib.rho`, which is compiled separately, contains:

```
id = x -> x;
```

Finally, a user program can then use `id` in the following way:

```
#include "stdlib.rh"
x = id 123;
```

4.2 Syntax

This section presents the syntax of the RhoStratego language. Figure 4.1 shows the syntax in BNF notation. Boldface and italics are used to denote terminals and non-terminals, respectively.

The language presented here is a subset of the actual RhoStratego language. Syntactic sugar such as infix and prefix notation for operators, infix notation for function

application, and transformational patterns are left out. Also left out are primitive types (integers and strings) and primitive operations ('primops', such as addition of integers and I/O). Treatment of the syntax and semantics for data type declarations and type annotations is postponed until chapter 6.

4.3 Semantics

This section presents the semantics of the RhoStratego language. We first present a set of rewrite rules on the language. By taking different subsets of these rules, we obtain lazy or strict semantics. Note that all set of rules as a whole gives a non-confluent calculus. For example, in a lazy semantics, `const 123 fail` will evaluate to `123`, but in a strict semantics, it evaluates to `fail` since arguments are evaluated first. However, the lazy and strict subsets themselves *are* confluent. For example, in the lazy semantics `const 123 fail` cannot evaluate to `fail`, even if we reduce the argument first, because the PropArg rule (given below) required here simply is not part of lazy semantics. So, unlike in the pure λ-calculus, lazy and strict versions of the language are not simply different reduction strategies; they are different calculi.

The fact that the semantics is given as a set of rewrite rules from the language to the language, i.e., as source-to-source transformation, means that they can be used directly in, e.g., an optimiser or an interpreter for the language. In fact, an interpreter based directly on these rewrite rules is given in appendix B.

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 term e is in *normal form* if no rewrite rules are applicable to e. What constitutes a normal form depends on the set of rewrite rules, i.e., whether we are using a lazy or strict semantics.

4.3.1 The rewrite rules

In the following, we let x range over the variables and C over the constructors.

- All the rewrite rules below assume that the left-hand side term has the form **let** ds **in** e. The idea is that the let-environment represents the memory, the 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 `lets`, we need the following trivial rule to lift these into the canonical form:

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

  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 the 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**.

- In a lazy semantics, if the body of a let is a let, we can merge the definitions, provided that there are no name clashes:

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

- In a strict semantics, things are not quite so simple. We have to verify that none of the definitions evaluates to `fail`. This is expressed by the following monstrosity.

$$\textsc{LetLet}^{+}: \frac{\begin{array}{l}\mathrm{defs}(ds_1) \cap \mathrm{defs}(ds_2) = \emptyset \\ \wedge\, \forall (x = e_2;) \in ds_2 :\ \textbf{let}\ ds_1 ds_2\ \textbf{in}\ e_2 \mapsto \textbf{let}\ ds'\ \textbf{in}\ e_2' \\ \quad \wedge\, e_2' \text{ is a normal form} \wedge e_2' \neq \textbf{fail}\end{array}}{\textbf{let}\ ds_1\ \textbf{in let}\ ds_2\ \textbf{in}\ e \mapsto \textbf{let}\ ds_1 ds_2\ \textbf{in}\ e}$$

$$\textsc{LetLet}^{-}: \frac{\begin{array}{l}\mathrm{defs}(ds_1) \cap \mathrm{defs}(ds_2) = \emptyset \\ \wedge\, \exists (x = e_2;) \in ds_2 :\ \textbf{let}\ ds_1 ds_2\ \textbf{in}\ e_2 \mapsto \textbf{let}\ ds'\ \textbf{in fail}\end{array}}{\textbf{let}\ ds_1\ \textbf{in let}\ ds_2\ \textbf{in}\ e \mapsto \textbf{let}\ ds_1 ds_2\ \textbf{in fail}}$$

- The following simple rule expresses that a variable may be substituted by its definition:

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

  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.

  There are even more elaborate variations:

$$\textsc{Var}: \frac{\begin{array}{l}x{=}e; \in ds\ \wedge\ e\ \neq\ \textbf{blackhole} \wedge \\ \textbf{let}\ ds\ \wr\ (x,\ \textbf{blackhole})\ \textbf{in}\ e \mapsto \textbf{let}\ ds'\ \textbf{in}\ e'\end{array}}{\textbf{let}\ ds\ \textbf{in}\ x \mapsto \textbf{let}\ ds'\ \wr\ (x,\ e')\ \textbf{in}\ e'}$$

  This rule implements a technique called *blackholing* [Pey92] to detect certain kinds of infinite recursion. Operationally, it means that when a closure is evaluated, it is updated with a special value called a black hole. Once evaluation of the closure is complete, it is updated again with the final result and the black hole disappears. If a black hole is ever entered, therefore, it necessarily means that we are trying to evaluate a closure that is already being evaluated — a non-terminating computation.

- The fundamental axiom of the λ-calculus, β-reduction, is expressed by means of *explicit substitution*: 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 function.

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

  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.

- Applying a function with a constructor pattern C to the constructor C will succeed:

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

  With this and the following match rules, it should be noted that the EvalArg rule (see below) can be used to bring the argument into the required normal form.

- On the other hand, applying a function with a constructor pattern C_1 to a different constructor C_2 will fail:

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

- 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.

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

- Otherwise, the application pattern match fails:

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

- Likewise, we can match against failure.

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

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

- The rules for matching against literals (e.g., integer constants) are omitted; they are very similar to the previous matching rules.

- If the left-hand side e_1 of a function application rewrites to e_1', we can replace e_1 by e_1':

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

- Likewise for the right-hand side of an application and the left-hand and right-hand sides of a choice:

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

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

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

  It should be noted that the EvalRight rule is not actually used in the lazy and strict evaluation strategies, since we only need to evaluate the right-hand side if the left-hand side fails, and in that case, the RChoice rule (given below) is applicable.

- We can choose the left-hand side of a choice if it 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.

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

- If the left-hand side of a choice is a cut expression, then the cut is removed. Note that only one cut is removed; this allows an expression to escape several choices by applying several cuts.

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

- We can choose the right-hand side of a choice if the left-hand side failed:

$$\textsc{RChoice} : \mathbf{let}\ ds\ \mathbf{in\ fail}\ \texttt{<+}\ e \mapsto \mathbf{let}\ ds\ \mathbf{in}\ e$$

- Applying failure to an expression yields failure; the following rule propagates failure in the function side of an application.

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

- Under a strict evaluation regime, the arguments to a function are evaluated first, so we need the following rule to propagate failure in the argument side of an application.

$$\textsc{PropArg} : \mathbf{let}\ ds\ \mathbf{in}\ e\ \mathbf{fail} \mapsto \mathbf{let}\ ds\ \mathbf{in\ fail}$$

- In certain contexts, we need to remove cuts from an expression; namely, cuts appearing in the function or argument sides of an application.

$$\textsc{UncutLeft} : \mathbf{let}\ ds\ \mathbf{in}\ \texttt{\^{}}e_1\ e_2 \mapsto \mathbf{let}\ ds\ \mathbf{in}\ e_1\ e_2$$

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

  A strict pattern is a pattern that forces evaluation of the argument, i.e., anything other than a variable.

- Finally, the following rule pushes arguments into a choice, as discussed in the previous section.

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

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

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

  Together with the alternative Var rule, this prevents e_3 from being evaluated more than once. Again, the fact that x does not occur in ds implies that it does not occur free in $e_{\{1,2,3\}}$.

The rules are summarised in figure 4.2.

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

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

$$\textsc{LetLet}^{+} : \frac{\begin{array}{l}\text{defs}(ds_1) \cap \text{defs}(ds_2) = \emptyset \\ \wedge\, \forall (x = e_2\textbf{;}) \in ds_2 : \textbf{ let } ds_1 ds_2 \textbf{ in } e_2 \mapsto \textbf{let } ds' \textbf{ in } e_2' \\ \quad \wedge\, e_2' \text{ is a normal form} \wedge e_2' \neq \textbf{fail}\end{array}}{\textbf{let } ds_1 \textbf{ in let } ds_2 \textbf{ in } e \mapsto \textbf{let } ds_1 ds_2 \textbf{ in } e}$$

$$\textsc{LetLet}^{-} : \frac{\begin{array}{l}\text{defs}(ds_1) \cap \text{defs}(ds_2) = \emptyset \\ \wedge\, \exists (x = e_2\textbf{;}) \in ds_2 : \textbf{ let } ds_1 ds_2 \textbf{ in } e_2 \mapsto \textbf{let } ds' \textbf{ in fail}\end{array}}{\textbf{let } ds_1 \textbf{ in let } ds_2 \textbf{ in } e \mapsto \textbf{let } ds_1 ds_2 \textbf{ in fail}}$$

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

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

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

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

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

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

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

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

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

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

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

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

Figure 4.2: RhoStratego evaluation rules

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

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

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

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

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

$$\textsc{UncutLeft} : \textbf{let } ds \textbf{ in } \hat{}e_1\ e_2 \mapsto \textbf{let } ds \textbf{ in } e_1\ e_2$$

$$\textsc{UncutRight} : \frac{p \text{ is a strict pattern}}{\textbf{let } ds \textbf{ in } (p \rightarrow e_1)\ \hat{}e_2 \mapsto \textbf{let } ds \textbf{ in } (p \rightarrow e_1)\ e_2}$$

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

Figure 4.2: Continued

4.3.2 Lazy evaluation strategy

We now give a lazy evaluation strategy for RhoStratego. Lazy languages are in some ways preferable to strict languages: just as languages with garbage collection free the programmer from the burden of having to specify explicitly the deallocation of objects, lazy evaluation frees the programmer from having to specify the evaluation order of values.

Reduction involves applying the rules in the previous subsection 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 following strategy E evaluates a term e lazily:

- If e is in normal form, we are done. A let-expression is in normal form if its body is a literal, a function, 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 function.
- Otherwise, do one of the following:
  - Apply the LetLet or Var rules.
  - Apply the EvalLeft rule with strategy E to normalise the left-hand side of a 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 get rid of any cuts, so we first apply the UncutLeft 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` $e_1 \ \ldots \ e_n$), which is a normal form. There are now two possibilities:

* We can apply exactly one of the Beta, Distrib, or PropFunc rules.
* Otherwise, we are looking at a strict pattern match: a match against a constructor, application, failure, or literal. This requires that we remove cuts from the argument, so UncutRight 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.

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

This evaluation strategy is formalised in appendix B.

4.3.3 Strict evaluation strategy

Strict evaluation differs from lazy evaluation in the following ways:

- The stop condition is the same, except that the notion of normal form is extended slightly: if the term is a constructor application, then all the arguments must also be in normal form.
- Instead of the LetLet rule we use LetLet$^+$ and LetLet$^-$, to evaluate all the definitions after adding them to the global let-environment.
- Evaluation of an application proceeds as follows:
  - Remove all cuts from the argument through use of the UncutRight rule. Then apply the EvalArg rule to evaluate the argument.
  - Remove all cuts from the function through use of the UncutLeft rule and evaluate it using the EvalFunc rule.
  - Try the following in order:
    * Use the FailMatch$^+$ or FailMatch$^-$ rules to perform a pattern match against `fail`. The only reason why we have to evaluate the function side at all is to expose `fail`-matches.
    * Use PropArg to propagate failure of the argument.
    * Otherwise the term is either a constructor application, in which case we are done (since its argument will have been evaluated by EvalArg), or we can apply one of the Beta, Distrib, PropFunc, or strict pattern match reduction rules.

The formal presentation of this strategy is given in appendix B as well.

4.4 Choice and 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 regular pattern matching is not

perfect [Tul00, EP00]. 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*, and *transformational patterns*. Furthermore, it makes existing sugar, such as Haskell's 'equational style' unnecessary (writing a function definition as a number of pattern-guarded equations which must be tried one after another), as well as `case`-expressions.

Case unnecessary Case-expressions, present in most functional languages, are unnecessary when we have a 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;
           _       -> 0;
```

can be written in RhoStratego as

```
f =    A       -> 123
    <+ B "foo" -> 456
    <+ _       -> 0;
```

Equational style unnecessary Haskell allows us to write patterns not just in λ-abstractors but also in function definitions:

$$\begin{array}{l}\texttt{f}\ p_{11}\ \ldots\ p_{1n}\ \texttt{=}\ e_1\\ \texttt{f}\ p_{21}\ \ldots\ p_{2n}\ \texttt{=}\ e_2\\ \ldots\\ \texttt{f}\ p_{m1}\ \ldots\ p_{mn}\ \texttt{=}\ e_m\end{array}$$

The semantics of these patterns is different from λ-abstraction: if a pattern match fails, the program does not diverge, but instead the next equation is tried. This is sugar for case-expressions that becomes redundant if we have a choice operator:

$$\begin{array}{l}\texttt{f =}\\ \quad p_{11}\ \ldots\ p_{1n}\ \rightarrow\ e_1\ \texttt{<+}\\ \quad p_{21}\ \ldots\ p_{2n}\ \rightarrow\ e_2\ \texttt{<+}\\ \quad \ldots\\ \quad p_{m1}\ \ldots\ p_{mn}\ \rightarrow\ e_m\end{array}$$

In fact, a major advantage is that pattern matching λ-abstractors are now first class, so we can write:

$$\begin{array}{l}\texttt{f}_1\ \texttt{=}\ p_{11}\ \ldots\ p_{1n}\ \rightarrow\ e_1\\ \texttt{f}_2\ \texttt{=}\ p_{21}\ \ldots\ p_{2n}\ \rightarrow\ e_2\\ \ldots\\ \texttt{f}_m\ \texttt{=}\ p_{m1}\ \ldots\ p_{mn}\ \rightarrow\ e_m\\ \texttt{f = f}_1\ \texttt{<+ f}_2\ \texttt{<+ ... <+ f}_m\end{array}$$

and we can combine the f_i arbitrarily.

Views Views [Wad87] 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 [BMS$^+$] 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 _) = 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 it, the previous example becomes (in RhoStratego):

```
f = reverse | ((y:_) -> 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`. 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 [EP00] redefine a guard as a list of qualifiers, just like 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 just write (in RhoStratego):

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

Alternatively, we could just get rid of the `Maybe` result of `lookup` altogether, making it of type `[(`α`, `β`)]` $\rightarrow$ α $\rightarrow$ β, and we arrive at:

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

Transformational patterns Transformational patterns [EP00] 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) <+ _ -> 0);
```

In general, a definition `f pat!fun = e` can be written as `f = fun | (pat → e)`. 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 _ -> x <+ {lin} -> 0;
```

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, _> -> x) (snoc y)
  <+ y -> (<> -> 0) (lin y);
```

First class patterns Tullsen [Tul00] 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.

4.5 Comparison to the ρ-calculus

RhoStratego's name derives from its origin as an experimental implementation of the *ρ-calculus*. The ρ-calculus, or *rewriting calculus*, developed in [CK98, Cir00] and simplified in [CKL01], 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 in [CKL01] as follows:

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

where x ranges over the variables and C ranges over the constants[2]. $t \rightarrow t$ represents abstraction, $t \bullet t$ stands for application, t, t builds a structure, and *null* denotes the

[2]This is slightly different from the convention in [CKL01], where the variables are $X, Y, Z, \ldots$ and the constants are $a, b, c, \ldots$.

empty structure. It is easy to see that the λ-calculus and term rewriting can be encoded in the ρ-calculus. For example, $x \to y \to x$ is exactly the λ-term $\lambda x.\lambda y.x$, and $(F(x) \to x) \bullet F(A)$ is the rewrite rule $F(x) \to x$ 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:

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

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.

Chapter 5

Compilation

5.1 Overview

The RhoStratego compiler translates a RhoStratego program to C code. The machine model is a tagless architecture [Pey92]. 'Tagless' means that all values (called closures), evaluated and unevaluated, have the same basic form, namely a code pointer that is *entered* to obtain its value. Once a closure has been evaluated, it is overwritten by a closure that contains the computed value and has a code pointer to a function that does nothing. Therefore a closure can be entered arbitrarily many times. In a tagged architecture, on the other hand, values are tagged to distinguish evaluated and unevaluated ones. Tagless architectures are advantageous because they can easily be adapted to support concurrency or distributed memory.

5.1.1 Machine model

The machine model has a garbage-collected **heap**. The heap consists of several kinds of objects:

- **Closures**. Following [Pey92], the term 'closure' is used here to refer to all values on the heap, evaluated or not. In other literature, 'closure' refers to suspensions and function values. There are two kinds of closures:
  - **Suspensions** (a.k.a. **thunks**). Suspensions contain all the information necessary to compute an expression into a value: namely, a code pointer and a way to access all the free variables occurring in the expression, through means of an environment (see below).
  - **Values**. Values are expressions that have been evaluated to normal form. The following kinds of values exist:
    - **Integer literals**. These store an integer in the machine's native form.
    - **String literals**. These contain a pointer to a null-terminated string.
    - **Constructors**. These contain a pointer to a statically allocated structure which is unique for each constructor. This is primarily so that its unique address can be used to distinguish constructors in pattern matches, but the structure can also store e.g. run-time type information.
    - **Constructor applications**. These contain two pointers to other closures; namely the left and right arguments of the application.

    - **Functions**. These contain a pointer to the code of the function and a pointer to the environment corresponding to the lexical scope of the function. A function value can be *called* by jumping to the function code, passing along the environment pointer and a pointer to the closure representing the argument.

- **Environments**. An environment corresponds to a lexical level in the (desugared and optimised) source program and contains pointers to all the closures added in the corresponding lexical level as well as a pointer to the enclosing environment (the *static link*). Each λ-abstraction that defines variables and each let-expression creates an environment. These must be on the heap, and not on the stack, since in a functional language a function may return a suspension, which contains a pointer to the environment; i.e. environments can outlive the function activation that created them. A smarter compiler can put some variables on the stack if it can prove that they are never used after the function that created them returns; but see the note below on the relative usefulness of stacks.

The machine also has a **stack**, which contains **activation records**. An activation record is created when a suspension is entered and when a function is called. It contains the return address of the caller, a pointer to the environment, and a pointer to the closure being evaluated (the one that should be updated with its normal form).

An alternative is to store activation records on the heap as well, and not use a call stack at all. Counterintuitively, systems that are entirely heap-based can be about as efficient as stack-based systems [AS96]. Putting activation records on the heap simplifies the implementation of choices, as discussed in section 5.2. Furthermore, garbage collection is greatly simplified; the only roots of the collector are the activation record of the caller that causes the garbage collection and the static closures (in the data segment). It may be advantageous to convert to continuation-passing style [Ste78] to simplify code generation in such a compiler.

The reason that we have not implemented this yet is that the C code generator becomes more complex since all details of setting up the activation records must be written out, whereas in the stack-based compiler this is done by the C compiler. This means, for example, that if function f makes a call to g, we must split f into two functions f_1 and f_2, where f_1 creates an activation record, using the address of f_2 as the return address, and jumps to g. When g completes, it will jump to the return address specified in its activation record, i.e., f_2. A complication is that all calls are tail calls, and it is not guaranteed that the C compiler will optimise those into tail jumps[1]. If the compiler does not do that, then the stack will never be contracted and we will very quickly run out of memory.

5.1.2 Compiler stages

The compiler accepts an ATerm representing the program. All expressions in the program have been decorated with their types as inferred by the type checker. The compiler produces a C file[2]. The compiler currently does not actually use any type information, so all type annotations are thrown out. In the future, the compiler could use this information to generate more efficient code.

These are the phases of the compiler:

[1] For example, `gcc` version 2.95 will not; the recently released version 3.0 will.

[2] To be precise, it produces an *Abox* term, which is a ATerm that can be fed into the pretty-printer `abox2text` to produce the actual C program text.

- **Pattern simplification**. In this phase, application match patterns and transformational patterns are removed. A simple pattern is:
  - A variable; or
  - A string or integer literal; or
  - A constructor; or
  - A `Decomp`-pattern. These are of the form `Decomp(x, y) -> e` where `x` and `y` are variables that are bound in `e`. The intended meaning is that such a function evaluates its argument to weak head-normal form and checks whether it is a constructor application. If so, `x` and `y` are bound to the left and right arguments of the application.

    `Decomp`-patterns are used to desugar application match patterns, using the following rule:

    $$p_1\ p_2\ \rightarrow\ e \mapsto \texttt{Decomp}(x_1, x_2)\ \rightarrow\ (p_1\ \rightarrow p_2\ \rightarrow e)\ x_1\ x_2$$

  Transformational patterns are just sugar for function application and matching against tuples (section 4.4).

- **Canonicalisation**. In this phase, the program is transformed so that each top-level definition is *simple*. A simple definition is either:
  - A λ-abstraction of which the left-hand side is a simple pattern; or
  - A let-expression in which all the definitions are simple *and* the body is an *atomised* expression.

  An atomised expression is:
  - A literal; or
  - A `fail`-expression; or
  - A primop; or
  - A choice operator, where both arguments are variables; or
  - A variable or constructor applied to zero or more variables; or
  - A cut of a variable.

- **Code generation**. Once all expressions are in canonical form, it is easy to generate code for them. The output of the compiler is C code, where the functions consist of calls to 'instructions' (actually macros or inlineable functions defined in a header file) of the abstract machine.

5.1.3 Abstract machine instructions

The operations of the abstract machine are listed below, with their semantics:

- `VARPAT()`. The initialiser for a function with a variable pattern, i.e., $x \rightarrow e$. Since such a function introduces a new variable x, a new environment must be created with a pointer to the closure of `x`.

- `DECOMPPAT()`. The initialiser for a function with a variable pattern, i.e., $\texttt{Decomp}(x, y) \rightarrow e$. Enter the argument and then check that it is an application. If so, create a new environment with two pointers to the left and right hand side of the application. If not, raise a failure.

- `INTPAT(n)`. The initialiser for a function with an integer pattern, i.e., $n \rightarrow e$. Enter the argument and check that it is an integer value equal to n. If that is not the case, raise a failure.
- `STRPAT(s)`. Idem for string patterns.
- `CONSTRPAT(constr)`. Idem for constructor patterns.
- `FAILPAT()`. The initialiser for a function with an failure pattern, i.e., `fail` $\rightarrow$ e. Install a failure handler and enter the argument. If failure does not occur, raise a failure. If failure does occur, catch it and proceed.
- `MKENV(size)`. Make a new current environment with the specified number of closure pointers. Set its static link to the previous current environment.
- `MKSUSP(disp, code)`. Allocate a new closure representing a suspension and store its pointer at slot `disp` in the current environment. This is used to initialise lets.
- `MKINT(res, n)` and `MKSTR(closure, s)`. Store an integer or string value in closure `res`, initialising it from an integer or string constant.
- `MKFUNC(res, env, funcptr)`. Store a function value in closure `res`, namely the function with the specified environment and function pointer.
- `MKCONSTR(res, constr)`. Store a constructor value in closure `res`; `constr` is the unique address of the constructor structure.
- `FAIL()`. Raise a failure (see section 5.2).
- `ENTER(closure)`. Enter the closure. When it returns, it has been updated with its normal form.
- `CALL(closure, arg)`. Call the closure, giving the closure `arg` as the argument.
- `COPY(dst, src)`. Copy the closure `src` to `dst`.
- `LCHOICE(res, left, right)`. Make a left-choice between closures `left` and `right`, storing the result in closure `res`.
- `CUT(res, arg)`. Evaluate closure `arg` and apply a cut to it (see section 5.2).
- `PRIMOP(name, res, env)`. Call the C function `name`, giving it the closure `res` where it should store its result, and the environment `env` that it can use to access its arguments.

5.1.4 Code generation

As stated above, it is very simple to generate code for canonicalised expressions. We generate code for a simple definition as follows. Compiling a function is very simple, since a function is a normal form; we emit one `MKFUNC()` instruction. The body is compiled into a separate function, the code of which is prefixed with the code for the simple pattern of the function. The other kind of simple definition, let-expressions, is also easy: the code consists of a `MKENV()`, a `MKSUSP()` for each definition, and the code for the body, which is an atomised expression. Each definition is compiled into a separate function.

Literals, `fail`-expressions, primops, choices, and cuts are trivial to compile. The only remaining kind of atomised expression, namely a variable or constructor applied

to zero or more variables, is compiled as follows. We emit a `ENTER()` or `MKCONSTR()` instruction for the left-most variable of constructor, and then produce a `CALL()` instructor for each argument.

The compiler also emits text to produce closures in the initialised data segment for top-level definitions, as well as constructor structures for all declared constructors[3].

5.1.5 Example

For example, the function

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

is compiled to

```
void const(Closure * res, Env * env) {
  MKFUNC(res, env, const_);
}
void const_(Closure * res, Env * env, Closure * arg) {
  VARPAT();
  MKFUNC(res, env, const__);
}
void const__(Closure * res, Env * env, Closure * arg) {
  VARPAT();
  ENTER(env->up->thks[0]);
  COPY(res, env->up->thks[0]);
}
```

Note that `const` itself is a normal form, so it just returns a function closure pointing to `const_`, which contains the actual code for `x -> y -> x`. `const_` is called when an argument is applied to the result of `const`. `const_` creates a new environment which a pointer to the argument and returns a function closure pointing to `const__`, which is the code for `y -> x`. When called, `const__`, adds *its* argument to the environment (redundantly, since `y` is never used), enters `x`, and copies the evaluated `x` into the result closure `res`. The expression `env->up->thks[0]` refers to the variable `x`.

The function

```
foo = C 123 -> "foo";
```

is first desugared into

```
foo = Decomp(x, y) -> (C -> 123 -> "foo") x y;
```

which is canonicalised into

```
foo =
  Decomp(x, y) ->
  let
    f = C -> 123 -> "foo";
  in f x y;
```

which is finally compiled to

[3]Since these may appear in all object files, they are declared as *weak* objects so that the linker collapses them into one.

```
void foo(Closure * res, Env * env) {
  MKFUNC(res, env, foo_);
}
void foo_(Closure * res, Env * env, Closure * arg) {
  DECOMPPAT();
  MKENV(1);
  MKSUSP(env, 0, c_0_2);
  ENTER(env->thks[0]);
  COPY(res, env->thks[0]);
  CALL(res, env->up->thks[0]);
  CALL(res, env->up->thks[1]);
}
void c_0_2(Closure * res, Env * env) {
  MKFUNC(res, env, c_0_2_);
}
void c_0_2_(Closure * res, Env * env, Closure * arg) {
  CONSTRPAT(c);
  MKFUNC(res, env, c_0_2__);
}
void c_0_2__(Closure * res, Env * env, Closure * arg) {
  INTPAT(123);
  MKSTR(res, "foo");
}
```

5.2 Implementation of the choice operator

To implement the choice operator, we must first decide how to represent failure values. Since any expression can evaluate to failure, a simple way to implement failure is to extend every value domain with a special value indicating failure. For example, in a function application, we would first evaluate the left-hand side and check whether it is a failure value; if so, we would return failure. In a left choice, we would evaluate the left-hand side, and if it returns failure, evaluate the right-hand side. This explicit tagging is wasteful in both space (the failure must be stored somewhere in the value) and time (we must constantly check for failure, even when failure has not occurred).

A much better method is to use the exception-handling techniques used in imperative languages (the choice operator is an exception-handling facility, after all). When an expression `x <+ y` is evaluated (entered), we set an exception handler and enter `x`. If `x` completes successfully, we remove the handler. If `x` does not complete successfully, on the other hand, i.e., if failure occurs, the handler is called, which then proceeds to enter `y`. The difficulty is that when failure occurs, we must restore the execution context that existed before we entered `x`, i.e., we must unwind the stack.

The machinery to do this exists in all C implementations through the `setjmp()` and `longjmp()` functions.. The function `setjmp()` saves the execution context in a structure; the function `longjmp()` restores the execution context saved in that structure. The context is restored so that it is exactly the same as at the time that `setjmp()` returned, *except* that the return value of `setjmp()` is non-zero. Here is an example:

```
jmp_buf context; /* execution context */

f()
```

```
{
  if (setjmp(&context) == 0) {
    /* Normal code path. */
    g(...);
    /* g() returned normally. */
  } else {
    /* g() called longjmp()! */
  }
}

g(...)
{
   ...
   if (...) longjmp(g, 1);
   ...
}
```

Using `setjmp()`/`longjmp()` we can implement the choice operator as follows:

- We maintain a stack of saved execution contexts. This is because choices may be nested. This stack is maintained separately from the call stack; this allows the execution context to be restored in $O(1)$ time in case of a failure.
- When a failure occurs, we pop the top execution context and restore it.
- When we enter `x <+ y`, we save the execution context on the top of the stack and enter `x`. If `x` completes successfully, we remove the saved execution context from the stack. If `x` fails, on the other hand, we enter `y`. The code for the left choice operator and fail therefore looks like this:

```
jmp_buf jmpbufs[...]; /* the stack */
unsigned int stacktop;

void LCHOICE(Closure * res, Closure * x, Closure * y)
{
  if (setjmp(jmpbufs[stacktop++] == 0)) {
    ENTER(x);     /* try x */
    stacktop--;   /* went okay */
    COPY(res, x);
  } else {
    /* x failed */
    ENTER(y);     /* try y instead */
    COPY(res, y);
  }
}

void FAIL()
{
  longjmp(jmpbufs[--stacktop], 1);
}
```

Distribution The above is not sufficient to implement RhoStratego's choice operator, however; if the left-hand side `x` returns a function, then, by the DISTRIB rule (section 4.3), `x` must be applied to the argument to `x <+ y`, *which we do not directly have access to*, and if that fails, `y` must be called and applied to the argument. Of

course, `x` may be a function of arbitrary arity, so we may need to accumulate any number of arguments.

We do this in the 'success' branch of `LCHOICE` (right after `stacktop--`). We check the result of `x`, and if it is a function, we create a 'fake' function (the *propagator*) which has a pointer to a special structure[4] that contains pointers to `x`, `y`, and all the arguments that have been applied so far (initially none).

Then, when the propagator is called with some argument `z`, it first calls the result of `x` with argument `z`. If that fails, we compute `y z`. If it is successful and the result is again a function, we create a new propagator that contains pointers to `x z`, `y`, and the arguments applied so far, namely `z`. This process repeats until the left-hand side fails or does not return a function.

Cuts The implementation of the choice operator must also deal with cuts. First, there is the issue of the representation of cuts. By the semantics, `^e` is transparent; that is, it automatically 'degrades' into `e` in all contexts where `e` is used, i.e., in pattern matches, primops, and applications of `e`.

One way to implement the cut is to wrap a dummy constructor around the value being cut; in other words, `^e` is translated to `C e` where `C` is a fixed, anonymous constructor known to the runtime system. This is very annoying, however, since it requires the cut(s) to be checked for and removed in all the aforementioned contexts. Since there may be any number of cuts, this requires a loop consisting of the steps (1) enter the closure; (2) if it is a `C e`, then go back to step (1) using closure `e`. The runtime cost of this is unacceptable. If, however, cuts were defined *not* to be transparent, then this approach would be sufficient.

A better solution, and the one used by the RhoStratego compiler, is to add to every function value a natural number indicating how many cuts have been applied to it. After all, cuts are only relevant to function values; all other values in the left-hand side of a choice cause the choice to return with that value, as is. So what we do is:

- When a function value is constructed, its counter is set to 0.
- The code for `^e` will enter `e` and make a copy of the resulting closure, and, if the result is a function, the counter in the copy will be increased by 1.
- The code for `x <+ y` will enter `x` and make a copy of the resulting closure, and, if the result is a function and its counter is greater than 0, will decrease the counter in the copy by 1.

In addition, failures can be cut, which is unrelated to functions being cut since failure is not a value in the runtime system. For example, `(^fail <+ 1) <+ 2` will evaluate to 2. To implement this properly, we do the following:

- Every time a failure is raised, we pass along a number indicating the number of cuts already applied to the failure. The expression `fail` has zero cuts applied to it, so the code for `fail` passes the value 0.
- The code for cut installs a failure handles and then evaluates its argument, removing the handler from the stack after successful completion of the argument. The failure handler will call the next failure handler *with the number of cuts increased by 1.*
- The failure handler for a choice will inspect the number of cuts, and if it is greater than 0, will call the next failure handler with the number of cuts

[4] It is a fake function because the pointer to the special structure masquerades as its environment pointer.

decreased by 1. If the number of cuts is 0, processing proceeds normally, i.e., the right-hand side of the choice is called.

This solution is still not complete. Consider, for example, the expression `((C -> 0) ^fail <+ 1) <+ 2`. By the semantics of the language, this should evaluate to `((C -> 0) fail <+ 1) <+ 2` ↦ `(fail <+ 1) <+ 2` ↦ `1 <+ 2` ↦ `1`. With the scheme described above, however, `fail` will throw a failure with cut count 0, which will be caught by the cut, which will increase the cut count to 1 and re-raise the failure, which will be caught by the failure handler of the inner choice. Since the cut count is 1, the right-hand side of the inner choice will not entered, but instead the failure will be re-raised, only to be caught by the failure handler of the outer choice; and the result will be `2`.

The problem here is that the cut has an effect only for values directly in the left-hand side of a choice. In the above example, `^fail` is merely an argument to the pattern match against `C -> 0`; and so `^fail` should degrade into `fail`, which will give a normal exception.

The solution is not very pleasant. We should ensure that an entered closure knows if its directly in the left-hand side of a choice. We can accomplish this by passing along a boolean value to that effect when we enter a closure. For example:

- The choice operator would call its left-hand argument with the flag set to `true`.
- In a copy expression of the form `x = y;` the closure for `y` is entered with the flag equal to the flag given to `x`.
- In an application, the left-hand side is entered with the flag set to `false`, but the actual call is made with the flag equal to the flag given to the application itself. For example, in `f x <+ y`, `f` is not directly under the choice, but the call `f x` *is*.
- Pattern matches and primops enter their arguments with the flag set to `false`. This ensures that e.g. `((C -> 0) ^fail <+ 1) <+ 2` works correctly.

5.3 Optimising choices

The choice operator allows us to write

```
f = (A -> ...) <+ (B -> ...) <+ (C x -> ...) <+ ...;
```

whereas in a more conventional language, such as Haskell we might write

```
f y = case y of { A -> ...; B -> ...; C x -> ...; ... }
```

As we have seen in section 4.4, the former is more powerful and convenient; on the other hand, the latter leads more naturally to efficient code. For example, the latter function would be compiled to something like this:

```
f: ENTER y
   if (constructor of y == A) then ...
   else if (constructor of y == B) then ...
   else if (constructor of y == C) then
     bind x to the first field of y in ...
   else ...
```

The RhoStratego compiler, on the other hand, produces inefficient code, involving lots of `setjmp()`s, etc.:

```
f:
  if (setjmp(...) == 0) {
    enter g [being the code for A -> ...]
  else
    enter f' [being the code for (B -> ...) <+ ...]

g:
  return a function g':

g'(arg):
  enter arg
  if (constructor of arg == A) then ... else FAIL()

f':
  if (setjmp(...) == 0) {
    enter h [being the code for B -> ...]
  else
    enter f'' [being the code for (C x -> ... <+ ...]
etc.
```

We conclude that `case`-statements, while redundant in the source language, *are* useful in the intermediate language. Assuming that we have a Haskell-like `case`-construct in the intermediate language, we can optimise the code by combining pattern matches in choices together `case`s.

The optimisation rules are:

$$\textsc{Func2Case} : p \to e \;\Rightarrow\; x \to \textbf{case}\ x\ \textbf{of}\ \{\ p \to e;\ \}$$

$$\textsc{Choice2Case} : \begin{array}{l} p \to e \texttt{ <+ } x \to \textbf{case}\ x\ \textbf{of}\ \{\ pats\ \} \;\Rightarrow \\ x \to \textbf{case}\ x\ \textbf{of}\ \{\ p \to e;\ pats\ \} \end{array}$$

Once we have rewritten appropriate choices into case-expressions, many well-known case-optimisations become available to us, such as the *case-of-known-constructor* (`case C of ... C -> ...`). These optimisations are especially useful in conjunction with inlining [PS98].

5.4 Implementation of generic traversals

The machine representation used here for constructed values is simple to implement, but inefficient: a constructor with n arguments will require $2n + 1$ closures (one closure for the constructor, n closures for the arguments, and n closures for the constructor applications)! In more conventional representations, the pointers to argument closures are stored in the constructor closure itself, so that only $n + 1$ closures are required (one for the constructor, and n for the arguments). There are two reasons why we cannot do that:

- In the untyped RhoStratego language, the arity of constructors is neither known at compile-time neither fixed at runtime (we can write a function that applies a constructor to an arbitrary number of arguments).

- In a application pattern match the left-hand side may be bound to a variable and applied to something else; hence, the left-hand side cannot be shared. This is also the case in the typed RhoStratego language.

Note that the space-complexity is the same: $\Theta(n)$.

If we want to use the conventional representation in the typed variant of the language, we can do so by making a copy of the closure in the application pattern match. But this is very inefficient, since deconstructing a term completely (as it happens in `all`, for example), will require $\Theta(n^2)$ space and time complexity (although the space will be garbage-collected).

Chapter 6

A type system for RhoStratego

6.1 Requirements

We place the following requirements on a type system for RhoStratego:

- The purpose of having a type system is that the programmer can express certain constraints about the values that can be passed to functions and the values than can be returned by functions, and have those constraints statically enforced by the compiler; i.e., we can guarantee at compile-time that the constraints will not be violated at runtime.
- Generic traversals such as `all` and `one` should be typeable. This entails typing application pattern matches.
- Flexibility. We want to be able to say things like

  ```
  f = collect (Var x -> x);
  ```

  to collect all the variables occurring in a term, regardless of term or type structure. In particular, we want to be able to express *type unifying* and *type preserving* generic transformations, i.e., those transformations that map an arbitrary term to some fixed type (like `collect` does) or the same type (like `all` does).

6.2 Overview

The basis of RhoStratego's type system is the Hindley-Milner type system [Mil78]. This system consists of the first-order typed λ-calculus extended with rank-1 parametric polymorphism.
We extend the Hindley-Milner system with algebraic data types, typing rules to support application pattern matches, and runtime type checks to support genericity.

The Hindley-Milner type system In the first-order typed λ-calculus, types are generated by the grammar $t ::= var \mid t \to t$. For example, the type of `id = x -> x` is $\alpha \to \alpha$. This is a monotype, however: any application of `id` will fix the type. The expression `id 3` will cause α to be substituted by the base type `Int`, giving `id` the type `Int -> Int`, and a subsequent expression `id "Foo"` will be rejected.

In order to obtain polymorphism, the second-order λ-calculus adds universal quantification, i.e., $t ::= var \mid t \to t \mid \forall var.t$. Unfortunately, while the second-order λ-calculus is quite powerful, type inferencing is undecidable [Wel99] — a serious drawback for practical use.

The Hindley-Milner type system, which we take as the basis of our type system, reduces the power of the second-order λ-calculus by generalising types only at the level of let-bindings, and specialising at each use of an identifier. Because of this all types are rank-1[1]. Type inferencing in this system *is* decidable.

Algebraic data types RhoStratego has algebraic data types, just like Haskell, with the important difference that data types and their constructors are declared separately. This gives us the ability to *extend* data types with new constructors, possibly in other modules, without losing type safety.

Example 29 Here are some examples of algebraic data types. Data types are declared using the keyword `data`. Data types can be parameterised with zero or more type variables. A constructors is declared by writing its type signature (the result of the type should be a declared data type).

```
data Bool;
True, False :: Bool;

data Point;
Point :: Int -> Int -> Point;

data Fork a;
Fork :: a -> a -> Fork a;

data Tup a b;
Tup :: a -> b -> Tup a b;

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

Note that specifying type variables in the data type declarations is a bit silly; we only need to know the *number* of type variables. Therefore, an alternative syntax would be to write `data Bool 0`, `data List 1`, `data Tup 2`, etc.

Type synonyms RhoStratego supports type synonyms, the equivalent of Haskell's `type` declarations. A type synonym is just syntactic sugar: all uses of a type synonym are replaced by its definition before typechecking commences. An example of a type synonym is:

```
syn IntList = List Int;
```

Note that the keyword `syn` is used instead of Haskell's keyword `type` to prevent confusion with data type declarations (`data`).

[1] The *rank* of a type is the maximum number $n+1$ such that there is a universal quantifier that is contained in n left branches of function types. For example, $\forall\alpha.(\alpha \to (\forall\beta.(\beta \to \alpha)))$ is a rank-1 type, and $(\forall\alpha.((\forall\beta.(\beta \to \beta)) \to \alpha)) \to$ `Int` is a rank-3 type.

Application pattern matches Certain generic traversal functions such as `all` and `one` are implemented using application pattern matches. How do we type this language construct?

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 that the definition of `all` reads:

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

The type of the right-hand alternative, `id`, is of course $\alpha \to \alpha$ (after instantiation). We define that the left and right arguments of a choice must have the same types. 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[2], `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., 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 more complicated than that; the exact typing rules for application pattern matches are given in the next section).

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)`. We now have the following types:

`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

From this we can derive:

`all f`	::	$\tau_3 \to \tau_3$	(after instantiating $\forall\beta$ with τ_3)
`all f c`	::	$\tau_2 \to \tau_1$	
`f x`	::	τ_4	(after instantiating $\forall\alpha$ with τ_4)
`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.

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 *runtime type check mechanism*. We write:

[2]The τ_i denote freshly instantiated type variables.

```
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 the function `collect` (defined in appendix D, which returns a *set* of subterms for which a certain function succeeds. For example, `varNames` collects the set of variables occuring in a term:

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

Applied to the term

```
App (App (Var "f") (App (Var "g") (Var "x"))) (Var "x")
```

(i.e., the representation of the program fragment `f (g x) x`), it will return the list `[f, g, x]`. The type of `collect` is $\forall\alpha.\forall\beta.(\forall\gamma.\gamma \to \beta) \to \alpha \to [\beta]$ (the argument function maps everything to some β). The type of `Exp?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.

Sum types The RhoStratego type system originally supported sum types, and each constructor had its own data type. For example, lists could be represented as:

```
Nil :: Nil;
Cons :: a -> List a -> Cons a;
syn List a = Nil + Cons a;
```

Then the expression `Nil` has not only type `Nil`, but also `Nil + Cons a`; in other words, a type `A` can be injected into `A + B`.

The problem with this approach is that the concept of type-preservation, which is essential for ensuring the type-safety of functions like `all`, becomes inexpressible.

t	$\rightarrow$	*var*	(type variable)
	$\mid$	$\forall$*var* . t	(universal quantification)
	$\mid$	*constr*	(constructor)
	$\mid$	$t\ t$	(application)
	$\mid$	$t \rightarrow t$	(function)
	$\mid$	**?**t	(guarded type)
	$\mid$	**Gen**(t^+)	(genericity)
decl	$\rightarrow$	**data** *constr var****;**	(data type declaration)
	$\mid$	*constr* **::** t**;**	(constructor declaration)

Figure 6.1: RhoStratego abstract type syntax

For example, is `A + B` $\rightarrow$ `A + B` a type-preserving function (and therefore, can it be given as an argument to `all`)? The answer must be 'no', since otherwise we can transform a value of type `A` into a value of type `B`.

For this reason we have opted for a more conventional, Haskell-like, system for data types, with the additional feature that it is extensible.

6.3 The type system

In this section we give a formal presentation of the type system of RhoStratego.

6.3.1 Syntax

The abstract syntax of types is giving in figure 6.1, which extends the grammar for RhoStratego given in 4.1. Syntactic sugar for lists, tuples, and type synonyms has been omitted.

6.3.2 Preliminaries

Definition 1 (Notation) In this section, we use x to indicate variables, α to indicate type variables, C to indicate constructors, e to indicate expressions, and τ and σ to indicate types.

Definition 2 (Environments) An environment (or *context*) Γ is a set of *type assignments* $x : \tau$, where x is a variable and τ a type.

Definition 3 (Free variables) The function fv that returns the set of type variables that occur free in a type τ is defined inductively as follows:

$$
\begin{array}{lcl}
\mathrm{fv}(\alpha) & = & \{\alpha\} \\
\mathrm{fv}(\forall\alpha.\tau) & = & \mathrm{fv}(\tau) - \{\alpha\} \\
\mathrm{fv}(C) & = & \emptyset \\
\mathrm{fv}(\tau_1\ \tau_2) & = & \mathrm{fv}(\tau_1) \cup \mathrm{fv}(\tau_2) \\
\mathrm{fv}(\tau_1 \rightarrow \tau_2) & = & \mathrm{fv}(\tau_1) \cup \mathrm{fv}(\tau_2) \\
\mathrm{fv}(?\tau) & = & \mathrm{fv}(\tau) \\
\mathrm{fv}(\mathbf{Gen}(\tau, \ldots)) & = & \mathrm{fv}(\tau)
\end{array}
$$

We also overload fv by defining $\mathrm{fv}(\Gamma)$ as the union of the sets of free variables occurring in the right-hand sides of the type assignments in Γ.

Definition 4 (Substitutions) The substitution $[\alpha := \sigma]$ applied to a type τ (notation $[\alpha := \sigma]\tau$) is defined inductively as follows:

$$
\begin{array}{lcl}
[\alpha := \sigma]\alpha & = & \sigma \\
[\alpha := \sigma]\alpha' & = & \alpha' \text{ if } \alpha \neq \alpha' \\
[\alpha := \sigma]C & = & C \\
[\alpha := \sigma](\tau_1\ \tau_2) & = & ([\alpha := \sigma]\tau_1)([\alpha := \sigma]\tau_2) \\
[\alpha := \sigma](\tau_1 \to \tau_2) & = & ([\alpha := \sigma]\tau_1) \to ([\alpha := \sigma]\tau_2) \\
[\alpha := \sigma](?\tau) & = & ?([\alpha := \sigma]\tau) \\
[\alpha := \sigma]\mathbf{Gen}(\tau_0, \ldots, \tau_n) & = & \mathbf{Gen}([\alpha := \sigma]\tau_0, \ldots, [\alpha := \sigma]\tau_n)
\end{array}
$$

6.3.3 Typing rules

The inference rules of the type system are given in figures 6.2 and 6.3, for terms and patterns respectively (the rule Abs requires that we can assign a type to a pattern). The trivial typing rules for integer and string literals have been omitted.

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

In the Abs and Let rules, Γ_p and Γ_{ds} refer to a 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 $\alpha \to \alpha$.

Application pattern matches 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 ensure that no substitutions are ever made to $\alpha_0 \ldots \alpha_n$; otherwise, the result will not be generic. For example, consider the following:

```
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.

Example 30 (Application pattern matches) Below is a natural deduction proof that the type of

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6.2: Typing rules for terms

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

$$\textsc{PCon} : \frac{\begin{array}{l}n \ge 0 \ \wedge \ C : (\sigma_1 \to \ldots \to \sigma_n \to \tau) \in \Gamma \ \wedge \\ \Gamma \vdash p_1 : \sigma_1 \ \wedge \ \ldots \ \wedge \ \Gamma \vdash p_n : \sigma_n\end{array}}{\Gamma \vdash (C\ p_1\ \ldots\ p_n) : \tau}$$

$$\textsc{Generic} : \frac{\begin{array}{l}n \ge 1 \ \wedge \ x_0 : (\alpha_1 \to \ldots \to \alpha_n \to \alpha_0) \in \Gamma \ \wedge \\ x_1 : \alpha_1 \in \Gamma \ldots \ \wedge \ x_n : \alpha_n \in \Gamma\end{array}}{\Gamma \vdash (x_0\ x_1\ \ldots\ x_n) : \mathbf{Gen}(\alpha_0, \alpha_1, \ldots, \alpha_n)}$$

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

Figure 6.3: Typing rules for patterns

is $\forall\beta.(\forall\alpha.\alpha \to \alpha) \to \beta \to \beta$. Ellipses are used to save space where necessary.

1.	`st` : $\forall\alpha.\forall\beta.(\alpha \to \beta) \to \alpha \to \beta$	ass.
2.	`id` : $\forall\alpha.\alpha \to \alpha$	ass.
3.	`all` : $\forall\beta.(\forall\alpha.\alpha \to \alpha) \to \beta \to \beta$	ass.
4.	`f` : $\forall\alpha.\alpha \to \alpha$	ass.
5.	`c` : $\tau_1 \to \tau_0$, `x` : τ_1	ass.
6.	`c x` : $\mathbf{Gen}(\tau_0, \tau_1)$	GENERIC, 5
7.	`all` : $(\forall\alpha.\alpha \to \alpha) \to (\tau_1 \to \tau_0) \to \tau_1 \to \tau_0$	INST, 3
8.	`all f` : $(\tau_1 \to \tau_0) \to \tau_1 \to \tau_0$	APP, 4, 7
9.	`all f c` : $\tau_1 \to \tau_0$	APP, 5, 8
10.	`st` : $\forall\beta.(\tau_1 \to \beta) \to \tau_1 \to \beta$	INST, 1
11.	`st` : $(\tau_1 \to \tau_0) \to \tau_1 \to \tau_0$	INST, 10
12.	`st (all f c)` : $\tau_1 \to \tau_0$	APP, 9, 11
13.	`f` : $\tau_1 \to \tau_1$	INST, 4
14.	`f x` : τ_1	APP, 5, 13
15.	`st (all f c) (f x)` : τ_0	APP, 12, 14
16.	`^(st (all f c) (f x))` : τ_0	CUT, 15
17.	`c x -> ...` : $\mathbf{Gen}(\tau_0, \tau_1) \to \tau_0$	ABS, 5, 6, 16
18.	`c x -> ...` : $\tau_0 \to \tau_0$	CONTRACT, 17
19.	`id` : $\tau_0 \to \tau_0$	INST, 2
20.	`c x -> ... <+ id` : $\tau_0 \to \tau_0$	CHOICE, 18, 19
21.	`f -> ...` : $(\forall\alpha.\alpha \to \alpha) \to \tau_0 \to \tau_0$	ABS, 4, 20
22.	`f -> ...` : $\forall\beta.(\forall\alpha.\alpha \to \alpha) \to \beta \to \beta$	GEN, 21

Example 31 (Runtime type check) Here is a proof that the type of

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

if $\forall\alpha.\alpha \to \alpha$. We assume the existence of the trivial STRING and PSTRING rules to type string literals in terms and patterns.

1.	`topdown` : $\forall\beta.(\forall\alpha.\alpha \to \alpha) \to \beta \to \beta$	ass.
2.	`try` : $\forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$	ass.
3.	`Var` : `String` $\to$ `Exp`	ass.
4.	`"x"` : `String`	PSTRING
5.	`Var "x"` : `Exp`	PCON, 3, 4
6.	`Exp?Var "x"` : `?Exp`	RTTC, 5
7.	`"y"` : `String`	STRING
8.	`Var "y"` : `Exp`	APP, 3, 7
9.	`Exp?Var "x" -> Var "y"` : `?Exp` $\to$ `Exp`	ABS, 6, 8
10.	`Exp?Var "x" -> Var "y"` : $\tau_0 \to \tau_0$	WIDEN, 9
11.	`try` : $(\tau_0 \to \tau_0) \to \tau_0 \to \tau_0$	INST, 2
12.	`try (Exp?Var "x" -> ...)` : $\tau_0 \to \tau_0$	APP, 10, 11
13.	`try (Exp?Var "x" -> ...)` : $\forall\alpha.\alpha \to \alpha$	GEN, 12
14.	`topdown` : $(\forall\alpha.\alpha \to \alpha) \to \tau_1 \to \tau_1$	INST, 1
15.	`topdown (try (...))` : $\tau_1 \to \tau_1$	APP, 13, 14
16.	`topdown (try (...))` : $\forall\alpha.\alpha \to \alpha$	GEN, 15

6.3.4 Type inference algorithm

Below we describe a type inference algorithm for RhoStratego. The inferencer does not infer all types: we must give type annotations for universally quantified function arguments (i.e., the variables that lead to types of rank 2 or higher). For example, we write `all` as follows:

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

Supertyping The notion of *supertyping* will be useful in the formulation of the inferencer. We often need to determine whether two types 'match' in a certain sense; for example, if a function has a formal argument of type τ, can we apply it to a value of type σ? This is slightly broader than syntactic unification. For example, a function accepting an argument of type $\forall\alpha.\alpha \to \alpha$ can be given a value of type `?Id -> Id`. Other places where this is useful is in verifying type signatures and type annotations (e.g., a function with inferred type $\forall\alpha.\alpha \to \alpha$ may be given a type signature `Int` $\to$ `Int`).

If we view a type as a set of values, then the *supertype function* determines, given two types t_1 and t_2, whether $t_2 \subseteq t_1$. Since supertyping takes the place of unification, it can discover substitutions to be applied to the global type environment. We therefore say that a type τ_1 is a supertype of τ_2 with substitution s, notation $\tau_1 \geq \tau_2 \Rightarrow s$.

Example 32 Examples of supertyping are:

`Int` $\to$ `Int`	$\geq$	$\alpha \to \alpha$	$\Rightarrow\ s = [\alpha := Int]$
`Int` $\to$ `String`	$\geq$	$\alpha \to \alpha$	is rejected
`Int` $\to$ `Int`	$\geq$	$\forall\alpha.\alpha \to \alpha$	$\Rightarrow\ s = []$
$\alpha \to \alpha$	$\geq$	$\forall\alpha.\alpha \to \alpha$	$\Rightarrow\ s = [\alpha := \tau_1]$
$\forall\alpha.\alpha \to \alpha$	$\geq$	?Id $\to$ Id	$\Rightarrow\ s = []$

In the above, τ_1 represents a fresh type variable.

We determine whether a type τ_1 is a supertype of τ_2 using the following tests, to be applied from top to bottom. If none of them apply, then τ_1 is not a supertype of τ_2. We write $s\tau$ to denote the substitution s applied to τ, and $\text{concat}(s_1, \ldots, s_i)$ to denote concatenation of substitutions.

The first few rules describe plain matching.

$C \geq C$	$\Rightarrow$	$[]$
$\alpha \geq \alpha$	$\Rightarrow$	$[]$
$\alpha \geq \beta$	$\Rightarrow$	$[\alpha := \beta, \beta := \alpha]$
$\alpha \geq \tau$	$\Rightarrow$	$[\alpha := \tau]$
$\tau \geq \alpha$	$\Rightarrow$	$[\alpha := \tau]$
$\tau_1\tau_2 \geq \tau_3\tau_4$	$\Rightarrow$	$\text{concat}(s_l, s_r)$
		where $\tau_1 \geq \tau_3 \Rightarrow s_l$
		$\wedge\ s_l\tau_2 \geq s_l\tau_4 \Rightarrow s_r$

The WIDEN typing rule is integrated into the supertype clauses for function types.

$\alpha \to \tau_2 \geq ?\tau_3 \to \tau_4$	$\Rightarrow$	s'
		where $\alpha \to \tau_2 \geq \tau_3 \to \tau_4 \Rightarrow s$
		$\wedge\ s' = s$ with subst. for α removed
$\tau_1 \to \tau_2 \geq ?\tau_3 \to \tau_4$	$\Rightarrow$	s'
		where $\tau_2 \geq \tau_4 \Rightarrow s$
$\tau_1 \to \tau_2 \geq \tau_3 \to \tau_4$	$\Rightarrow$	$\text{concat}(s_l, s_r)$
		where $\tau_3 \geq \tau_1 \Rightarrow s_l$
		$\wedge\ s_l\tau_2 \geq s_l\tau_4 \Rightarrow s_r$

The final clauses deal with universal quantification.

$$\begin{array}{lcl} \forall\alpha.\tau_1 \geq \forall\beta.\tau_2 & \Rightarrow & s \\ & & \text{where } \tau_1 \geq [\beta := \alpha]\tau_2 \Rightarrow s_l \\ & & \wedge \text{ s has no subst. for } \alpha \\ \tau_1 \geq \forall\alpha.\tau_2 & \Rightarrow & s \\ & & \text{where } \tau_1 \geq \text{rename}(\tau_2) \Rightarrow s \\ \forall\alpha.\tau_1 \geq \tau_2 & \Rightarrow & s \\ & & \text{where } \tau_1 \geq \tau_2 \Rightarrow s \\ & & \wedge \text{ s has no subst. for } \alpha \end{array}$$

The supertype function can easily be extended to support sum types. This was in fact the primary reason to implement type matching in this way, since we have to deal with the fact that, e.g., a function of type `A + B + C -> D` can be given an argument of type `A + C`, but not `A + E`. Sum types were subsequently removed, but the supertype function is still useful to such things as verifying type signatures and dealing with runtime type checks (i.e., the WIDEN rule).

$$\begin{array}{lcl} \sum_{1\leq i\leq n}\tau_{1,i} \geq \sum_{1\leq j\leq m}\tau_{2,j} & \Rightarrow & \text{concat}(s_j) \\ & & \text{where } \forall j.\exists i.\tau_{1,i} \geq \tau_{2,j} \Rightarrow s_j \end{array}$$

That is, each type in the right-hand sum should be a subtype of some type in the left-hand sum; for example, `(Int` $\rightarrow$ `Int) + B + C` $\geq$ `C +` $\forall\alpha.\alpha \rightarrow \alpha$, but not vice versa.

Inference algorithm For each kind of term in the language, the type is computed as follows:

- For **literals** (integers and strings) the type is fixed, e.g. `Int`.
- For a **failure**, return τ, where τ is a fresh type variable (the FAIL rule).
- For a **variable** x, take the type τ of x as specified in the environment (the VAR rule) and specialise it. Idem for constructors (the CON rule).
- For a **cut** `^e`, return the type of `e`; cuts do not affect the type of their argument (the CUT rule).
- For an **application** `(`e_1 e_2`)`, do the following:
  - Compute the type τ_1 of e_1.
  - Add to Γ the bindings $x := \alpha$ and $y := \beta$, where x and y are fresh variables (the names are irrelevant in any case) and α and β are fresh type variables.
  - Determine whether $\alpha \rightarrow \beta$ is a supertype of τ_1, and apply the resulting substitution to Γ. This tells us whether τ_1 is a function; if so, the type of its formal argument and result are bound to x and y in Γ, respectively.
  - Compute the type τ_2 of e_2.
  - Check that the type bound to α (i.e., the type of the formal argument) is a supertype of τ_2. If this fails, generalise the argument type τ_2 and try again. This is necessary to support rank-2 polymorphism. Apply substitutions to Γ.
  - Remove x and y from the environment and return the type bound to y (the type of the result, after substitutions).

- For a **choice** e_1 `<+` e_2, compute the types τ_1 and τ_2 of e_1 and e_2 respectively. Then either τ_1 should be a supertype of τ_2 or τ_2 a supertype of τ_1. The resulting type is the *largest* (or *most specific*) of the two, after substitutions. For example, the type of both `(1 -> 2) <+ id` and `id <+ (1 -> 2)` is `Int` $\rightarrow$ `Int`, not $\forall\alpha.\alpha \rightarrow \alpha$.

- For a **type annotation** e `::` τ, compute the type σ of `e` and determine whether τ is a supertype of σ (since a type annotation can widen the type of a value; we can write `id :: (Int -> Int)` but not `(1 :: (Int -> Int)) ::` $\forall\alpha.\alpha$. Apply the resulting substitution to Γ and τ and return τ.

- For a **rule** $p \rightarrow e$, do the following:

  - Compute the type σ of p, and the variables occurring therein. This can be done straight-forwardly using the pattern typing rules. Add the pattern variables to the environment, as well as a dummy binding `x :=` σ, so that substitutions discovered along the way are also applied to σ.
  - Compute the type τ of e.
  - Restore the environment. The type of $p \rightarrow e$ is $\sigma \rightarrow \tau$ (with substitutions applied).

- For a **let-expression**, do the following:

  - Formulate the *local environment*. This is the set of all locally declared type signatures, added to which is, for each definition $x = e$ that does not have a type signature, a signature $x :: \alpha$, where α is a fresh type variable. Add the local environment to Γ.
  - Compute the type of each definition, threading Γ through each type computation. For each definition:
    - Compute the type τ of the body.
    - Generalise τ and verify that it is a subtype of the type signature. For example, the definition `f = x` $\rightarrow$ `x` may be given a signature `f :: Int` $\rightarrow$ `Int`.
  - Compute the type of the body. This is the type of the entire let-expression.
  - Remove the local environment from Γ.

- Finally, **primops** are assigned an arbitrary type α, where α is a fresh type variable. After all, we can say nothing about the types of values produced externally. The result of a primop can be constrained using a type annotation.

An implementation of this algorithm is given in appendix C.

Chapter 7

Applications

In this chapter we present an application of RhoStratego: an inliner for RhoStratego. This chapter *is* in fact the RhoStratego program; the LaTeX source was generated from the annotated program text.

Note that inlining *in itself* is not a particularly useful optimisation for lazy languages; due to laziness, many definitions will end up as separate closures even when they are inlined. The real usefulness is in combination with other optimisations, such as β-reduction [PS98].

The first thing most RhoStratego programs do is include the header file to the standard library.

```
#include "stdlib.rh"
```

The header file `rhosyntax.rh` defines the abstract data type of RhoStratego.

```
#include "rhosyntax.rh"
```

Definitions should not be inlined indiscriminately, since that will result in code bloat. Only definitions below a certain size should be inlined. `termSize` implements a simple heuristic — the number of syntax nodes — to determine whether to inline.

```
termSize :: a . a -> Int;
termSize = c x -> termSize c + termSize x <+ x -> 1;
```

β-reduction is a form of inlining, and inlining often creates opportunities for β-reduction. Rather than treating it separately, we use explicit substitution to rewrite an application of a λ-abstraction into a let-expression, and let the general inliner take care of it.

The rule `appInwards` is used to float applications into let-expressions, in order to expose more opportunities for inlining. For example, `(x -> y -> x) 1 2` will be rewritten using `betaReduce'` into `(let x = 1; in y -> x) 2`, then using `appInwards` into `let x = 1; in (y -> x) 2`, and finally using another application of `betaReduce'` into `let x = 1; y = 1; in y -> x`.

```
betaReduce = repeat (oncetd (betaReduce' <+ appInwards));

betaReduce' = Term?App (Rule (Var x) e1) e2 -> Let [Def x e2] e1;

appInwards = Term?App (Let ds e1) e2 -> Let ds (App e1 e2);
```

`fvars` returns the list of free variables occurring in an expression.

```
fvars :: a . a -> [Id];
fvars =
  Term?Var x -> [x] <+
  Term?Rule p e -> diff (fvars e) (fvars p) <+
  Term?Let ds e -> diff (map (Def x _ -> x) ds) (letVars ds e) <+
  unions . (mapkids fvars);

letVars = ds -> e ->
  concat ((fvars e) : (map (Def _ e -> fvars e) ds));
```

`filterVars` filters variables from a list of bindings. A binding is a (name, term) tuple.

```
filterVars = vars -> filter (<x, _> -> not (x `elem` vars));
```

The function `inline` performs inlining of let-bindings. We traverse the term in a top-down fashion, passing down the set of let-bindings currently in scope.

```
inline = inline' [];

inline' :: a . [<Id, Term>] -> a -> a;
inline' = defs -> (
  let

    doProgram, doLet, doRule, doVar :: a . a -> a;
```

Treat programs as let-expressions for the purpose of inlining.

```
    doProgram = Program?Program ds ->
      (Let ds' _ -> Program ds') (doLet (Let ds Fail));
```

Let-expressions add bindings to the set.

```
    doLet = Term?Let ds e -> (
      let newdefs = map (Def x e'' -> <x, e''>) ds;
```

We only add those that are smaller than a certain size that should be determined empirically.

```
          newdefs' = filter (<x, e> -> termSize e < 10) newdefs;
          defs' = conc newdefs' defs;
```

When performing inlining inside a let-binding `x = e`, we should remove `x` from the set of bindings; otherwise, an infinite expansion could occur if `x` is recursive.

```
          ds' = map (Def x e -> inline'
            (filterVars [x] defs') (Def x e)) ds;
          e' = inline' defs' e;
      in Let ds' e');

    doRule = Term?Rule p e ->
      Rule p (inline' (filterVars (fvars p) defs) e);
```

This is where the actual inlining happens. We look up `x` in the set of bindings and replace it with its definition. Then we recursively inline the expansion, since there may be further opportunities for inlining. Note that `lookup`, and therefore `doVar`, fails if `x` is not in the set of bindings.

```
    doVar = Term?Var x -> st (inline' defs) (lookup x defs);
```

Try all the rules above; if they all fail, perform inlining on the subterms.

```
  in doProgram <+ doLet <+ doRule <+ doVar <+ all (inline' defs));
```

`removeDead` removes dead let-bindings, i.e., definitions not used anywhere in the let-expression. This could be a bit smarter; what we really care about is reachability of a definition from the body of the let (cf. the garbage collector in B.3).

```
removeDead, removeDead' :: a . a -> a;
removeDead =
  (Term?Let ds e -> removeDead' (
    let vars = letVars ds e;
        ds2 = filter (Def x _ -> x `elem` vars) ds;
    in Let ds2 e)) :: (a -> a)
  <+ removeDead';
removeDead' =
  all removeDead;
```

`emptyLet` removes all empty let-expressions (i.e., it's the opposite of the LetLift rule in section 4.3).

```
emptyLet = bottomup (try (Term?Let [] e -> e));
```

`optimise` is a pipeline consisting of the transformations defined above.

```
optimise = betaReduce | (inline | (removeDead | emptyLet));
```

The function `main` performs I/O. It reads a program in ATerm format, transforms it, and writes it out as an ATerm.

```
main =
  let a1 = readTerm "inlinertest.trm";
      a2 = writeTerm "" ATFmtText;
  in a1 >>= (x -> a2 (optimise x));
```

Chapter 8

Conclusion

We have implemented a lazy functional language integrating a number of interesting features from Stratego. From this project we can conclude the following:

- Application pattern matches are a simple but quite powerful primitive for constructing generic traversals (section 4.1).
- Choices liberate pattern matching; they subsume a large number of existing language constructs, including cases, the equational style, and views (section 4.4).
- Cuts are hard to implement if they are type-transparent, but easy if they are not (section 5.2).
- 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 good solution is to keep the language lazy, but make constructor applications strict. The (severe) disadvantage is that we lose the ability to make infinite or cyclic data structures; we keep the advantage of laziness of let-bindings and function arguments (which implies the ability to define control structures such as `if`). A better alternative might be to allow the programmer to indicate that certain constructor arguments are strict (as is possible in Haskell).

Future work

- It might be interesting to retarget the compiler from C to C--[PRR99, RP00]. At the very least this will give us tail jumps, without which many computations cannot be done in bounded space.
- The ability to read/write ATerms from certain Haskell implementations (e.g. GHC), i.e., integration into XT, could be useful for program transformation applications.
- Better control over the scope of the choice operator, 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 [PRHM99]. This is because different subexpressions may raise different exceptions, and so the exception that is actually raised depends on which subexpression is evaluated first.

- **Non-local variable bindings**. In Stratego we can write things like:

  ```
  fetch(?Foo(x)); !x
  ```

  That is, walk over a list until an element is encountered that matches with the pattern `Foo(x)`; then replace the current term with `x`. The interesting thing is that `x` is (implicitly) declared in the scope of the entire strategy, but it is defined (given a value) as a side-effect of executing `fetch`. Is something similar possible in a function language such as RhoStratego? Note that if `x` is defined (i.e., matched) multiple times, the various definitions must be equal (such as in `map(?Foo(x))` or `?Foo(x, x)`. A possible functional implementation would be:

  - A variable `x` is initially said to be *undefined*.
  - When a definition occurs and `x` is undefined, `x` gets the value of the definition.
  - When a definition occurs and `x` is defined, the new definition is evaluated and must be equal to the current one.
  - When a currently undefined variable `x` is evaluated, all expressions that contain definitions are evaluated. If `x` is still undefined afterwards, failure occurs.

- Is there a more elegant and/or efficient way to implement the cut operator than described in section 5.2? Likewise, is there a better implementation for RhoStratego's generic traversals than described in section 5.4?

Appendix A

Overview of the implementation of RhoStratego

We have implemented an interpreter and compiler for the RhoStratego language described in chapter 4, as well as an implementation of the type checker given in chapter 6.

The RhoStratego implementation consists of the following components:

- A parser for the RhoStratego language, generated from a grammar specified using the Syntax Definition Formalism (SDF2) [Vis97]. The grammar is translated to a parse table using the `sdf2table` program, which is part of XT. This table can then be used with the `sglr` and `implode-asfix` programs — also part of XT — to parse a file into an ATerm.
- A desugarer that takes the output of the parser and simplifies it.
- A pretty-printer, to transform RhoStratego abstract syntax to concrete syntax.
- An interpreter written in Stratego, implementing both the lazy and strict evaluation strategies. The interpreter is quite limited — it does not implement most I/O functions, for example — and is primarily useful for language experiments. The source code of the interpreter is given in appendix B.
- A type checker written in Stratego. It reads a desugared RhoStratego module in ATerm format, infers (as far as possible) and checks types, and produces a desugared RhoStratego module in ATerm format in which every expression has been decorated with its type.
- A compiler written in Stratego. It translates desugared RhoStratego modules in ATerm format to C code.
- A runtime system and standard library against which RhoStratego programs must be linked.
- A set of test cases (i.e., RhoStratego programs along with their expected outputs) to test the compiler against automatically.

The source code of the RhoStratego implementation can be obtained through anonymous CVS. On Unix systems this can be accomplished using the following commands:

```
% export CVSROOT=\
 :pserver:anoncvs@losser.st-lab.cs.uu.nl:/home/cvsrepo
% cvs login
[Password: anon]
% cvs -z9 checkout rho
```

A web interface to the CVS repository is available at `http://losser.st-lab.cs.uu.nl/services/cvsweb/rho/`. Further information can be obtained at `http://www.stratego-language.org/rho/`.

Appendix B

The RhoStratego interpreter

B.1 rho-laws.r

The module `rho-laws` defines the semantic rules of the RhoStratego language, as defined in section 4.3.

```
module rho-laws
imports lib rho rho-bindings rho-utils

rules
```

The following rules corresponding rather directly to those in section 4.3.

```
LetLift: e -> Let([], e)

LetLet:
  Let(ds1, Let(ds2, e)) -> Let(<conc> (ds2', ds1), e')
  where <rho-rename> Let(ds2, e) => Let(ds2', e')
```

LetLet' is the most complex rule by far. It deals with let-expressions in the strict evaluator. We add all definitions to the global environment after renaming (just like the LetLet rule does), and then we evaluate each definition in the order in which they appear in the let-expression. If any of them fails, the entire expression fails.

Some rules, like this one, are parameterised with a strategy determining evaluation of the subterms.

```
LetLet'(e):
  Let(ds1, Let(ds2, e)) -> e''
  where
    <rho-rename> Let(ds2, e) => Let(ds2', e');
    <conc> (ds2', ds1) => ds';
    <foldl(
      \(_, Let(ds, Fail)) -> Let(ds, Fail)\ <+
      Eval1Let(e)
    )> (ds2', Let(ds', e')) => e''

Eval1Let(e):
  (Def(x, _), Let(ds, e)) -> Let(ds', e')
```

```
  where
    <e> Let(ds, Var(x)) => Let(ds', e2);
    <Fail <+ !e> e2 => e'

LetVar:
  Let(ds, Var(x)) -> Let(ds, <LookupVar> (x, ds))

LetVar(e):
  Let(ds, Var(x)) -> Let(ds'', e')
  where <debug(!"eval ")> x;
        <LookupVar> (x, ds) => e;
        <e> Let(ds, e) => Let(ds', e');
        <map(try(?Def(x, _); !Def(x, e')))> ds' => ds''

Beta:
  Let(ds, App(Rule(Var(x), e1), e2)) ->
  Let(ds, Let([Def(x, e2)], e1))

ConMatchP:
  Let(ds, App(Rule(Con(c), e), Con(c))) -> Let(ds, e)

ConMatchN:
  Let(ds, App(Rule(Con(c), _), e2)) -> Let(ds, Fail)
  where <is-nf; not(?Con(c))> e2

IntMatchP:
  Let(ds, App(Rule(Const(Int(n)), e), Const(Int(n)))) ->
  Let(ds, e)

IntMatchN:
  Let(ds, App(Rule(Const(Int(n)), _), e2)) -> Let(ds, Fail)
  where <is-nf; not(?Const(Int(n)))> e2

AppMatchP:
  Let(ds, App(Rule(App(p1, p2), e3), App(e1, e2))) ->
  Let(ds, App(App(Rule(p1, Rule(p2, e3)), e1), e2))
  where <is-nf> App(e1, e2)

AppMatchN:
  Let(ds, App(Rule(App(p1, p2), _), e2)) -> Let(ds, Fail)
  where <is-nf; not(?App(_, _))> e2

FailMatchP:
  Let(ds, App(Rule(Fail, e), Fail)) -> Let(ds, e)

FailMatchN:
  Let(ds, App(Rule(Fail, _), e2)) -> Let(ds, Fail)
  where <is-nf; not(Fail)> e2

EvalFunc(e):
  Let(ds, App(e1, e2)) -> Let(ds', App(e1', e2))
  where <e> Let(ds, e1) => Let(ds', e1')

EvalArg(e):
```

```
    Let(ds, App(e1, e2)) -> Let(ds', App(e1, e2'))
    where <e> Let(ds, e2) => Let(ds', e2')

  EvalLeft(e):
    Let(ds, LChoice(e1, e2)) -> Let(ds', LChoice(e1', e2))
    where <e> Let(ds, e1) => Let(ds', e1')

  EvalRight(e):
    Let(ds, LChoice(e1, e2)) -> Let(ds', LChoice(e1, e2'))
    where <e> Let(ds, e2) => Let(ds', e2')

  LeftChoice:
    Let(ds, LChoice(e1, _)) -> Let(ds, e1)
    where <is-nf; not(Fail + Cut(id) + Rule(id, id))> e1

  LeftChoiceCut:
    Let(ds, LChoice(Cut(e1), _)) -> Let(ds, e1)

  RightChoice:
    Let(ds, LChoice(Fail, e)) -> Let(ds, e)

  PropFunc:
    Let(ds, App(Fail, _)) -> Let(ds, Fail)

  PropArg:
    Let(ds, App(_, Fail)) -> Let(ds, Fail)

  UncutLeft:
    Let(ds, App(Cut(e1), e2)) -> Let(ds, App(e1, e2))

  UncutRight:
    Let(ds, App(e1, Cut(e2))) -> Let(ds, App(e1, e2))

  Distrib:
    Let(ds, App(LChoice(e1, e2), e3)) ->
    Let(ds, Let([Def(x, e3)],
      LChoice(App(e1, Var(x)), App(e2, Var(x)))))
    where !Id("_x", <unique-int> ()) => x

strategies
```

`is-con` and `is-nf` determine whether a term is a constructor application or a normal form, respectively.

```
  is-con = rec x(Con(id) + App(x, id))

  is-nf =
    Const(id) + Rule(id, id)
    + is-con + Fail + Cut(id)
    + LChoice(Rule(id, id), id)
```

B.2 rho-bindings.r

The module `rho-bindings` defines strategies for renaming of and substitutions on RhoStratego terms and types, using the language-independent generic strategies defined in [Vis00].

```
module rho-bindings
imports lib substitution rho
```

First we define renaming and substitutions for terms. This is done by the generic `rename` and `substitute` strategies, which must be parameterised with a number of strategies:

- `AtVar(s)` that applies some strategy to an identifier (such as replacing it with a new name).
- `RhoBound` that returns the variables defined in an expression (i.e., the variables in the left-hand side of a function and those defined in a let-expression).
- `RhoBoundIn` indicates *where* variables are bound. This is slightly complicated by the fact that variables defined in transformational patterns are not bound in the right-hand side of the function in which the pattern appears.
- `RhoPaste` replaces variables at binding sites.
- `RhoNewVar` generates a new identifier, given an existing identifier. In the RhoStratego interpreter and compiler, identifiers are not simply strings but (name, number) tuples. This is to preserve the original name when renaming, for more intelligible output or error messages. When we rename an identifier, the name stays the same, but the number is a new unique number.

```
rules

  IsVar: Var(x) -> [Var(x)]

  AtVar(s): Var(x) -> Var(<s> Var(x))

  RhoBound: Rule(t1, t2) -> <collect(Var(id))> t1

  RhoBound: Let(defs, t) ->
    <map(\Def(Id(nm, n), t) -> Var(Id(nm, n))\ )> defs

  RhoNewVar: Var(Id(name, nr)) -> Id(name, <unique-int> ())

strategies

  boundin-pat(bnd,ubnd) =
    rec x(XPat(ubnd) + Var(id); bnd + App(x, x) + Const(id))

  RhoBoundIn(bnd, ubnd, ignore) =
    Rule(boundin-pat(bnd, ubnd), bnd) +
    Let(bnd, bnd)

  RhoPaste(nwvars) =
    Rule(id, id) +
```

```
    Let(split(nwvars, id);
        zip(\ (x, Def(_, t)) -> Def(x, t)\ )
       ,id)

  rho-substitute = substitute(
    Var(id), AtVar, RhoBound, RhoBoundIn, RhoPaste)

  rho-free-vars = free-vars(IsVar, RhoBound)

  rho-rename = rename(
    AtVar, RhoBound, RhoBoundIn, RhoPaste, RhoNewVar)
```

Similarly, we define renaming and substitutions for types.

```
rules

  IsTVar: TVar(x) -> [TVar(x)]

  AtTVar(s): TVar(x) -> TVar(<s> x)

  RhoTBound: TForAll(var, tp) -> [var]

strategies

  RhoTBoundIn(bnd, ubnd, ignore) = TForAll(bnd, bnd)

  RhoTPaste(nwvars) = TForAll(nwvars; \[x] -> x\, id)

  rho-free-tvars = free-vars(IsTVar, RhoTBound)

  rho-trename = rename(
    AtTVar, RhoTBound, RhoTBoundIn, RhoTPaste)
```

The generic renamer defined in Stratego's standard library is not entirely generic enough: it doesn't allow us to define how new variable names are generated — it just generates a random new string not currently in use. However, we want to retain the original name of a variable when we rename it, in the manner described above. Therefore the following renamer is parameterised with a strategy defining how new identifiers are to be generated.

```
rules

  RnBinding(bndvrs, paste, newvar) :
    (t, env1) -> (<paste(!ys)> t, env1, env2)
    where <bndvrs> t => xs; map(newvar) => ys;
          <conc>(<zip(id)>(xs,ys), env1) => env2

strategies

  rename(isvar, bndvars, boundin, paste, newvar) =
    \ t -> (t, []) \ ;
    rec x(env-alltd(RnVar(isvar)
                    <+ RnBinding(bndvars, paste, newvar);
                       DistBinding(x, boundin)))
```

```
unique-int = prim("uniqueInt")
```

B.3 rho-interpreter.r

The module `rho-interpreter` defines lazy and strict evaluation strategies for RhoStratego (as defined in sections 4.3.2 and 4.3.3, respectively), using the rewrite rules given in `rho-laws`.

```
module rho-interpreter
imports lib rho rho-laws pack-graph

strategies
```

`rho-eval-lazy-io` is the "main" function of the lazy interpreter. We evaluate a program by first translating it to a let-expression consisting of all top-level declaration, and the application of the standard function `force` to the function `main` defined in the input program. `force` ensures that the resulting term is evaluated completely; this is in effect Haskell's `show` function.

```
  rho-eval-lazy-io = iowrap(
    ?Program(decls);
    !Let(decls, App(Var(Id("force", 0)), Var(Id("main", 0))));
    remove-hastypes;
    collect-garbage;
    rho-eval-lazy;
    ?Let(decls', result);
```

Even though force forces strictness of the result, there can be some indirection, e.g. `let x = 123; in C x`. So we replace all variables with their definition.

```
    where (<topdown(repeat(\Var(x) ->
      <LookupVar> (x, decls')\))> result => result');
    !result'
  )
```

This is the lazy evaluator, as explained in section 4.3.2. Note how we have a clean separation between the rewrite rules that do the actual work, and the strategy that controls how the rewrite rules are applied.

```
  rho-eval-lazy = rec e(
    Let(id, is-nf)
    + ( LetLet
//    + LetVar // non-sharing version
      + LetVar(e)
      + EvalLeft(e);
        (LeftChoice + LeftChoiceCut + RightChoice)
      + repeat(UncutLeft);
        EvalFunc(e);
        ( Beta
        + Distrib
        + PropFunc
        + Let(id, is-con)
```

```
      <+ repeat(UncutRight);
        EvalArg(e);
        ( ConMatchP + ConMatchN
        + IntMatchP + IntMatchN
        + AppMatchP + AppMatchN
        + FailMatchP + FailMatchN
        )
      )
    <+ \x -> <fatal-error> ["got stuck in term ", x]\
    ); e
  )
```

`rho-eval-lazy-io` is the main function of the strict interpreter. This is a bit simpler than the wrapper around the lazy evaluator, since we don't have to force strictness of the result.

```
rho-eval-strict-io = iowrap(
  ?Program(decls);
  !Let(decls, Var(Id("main", 0)));
  remove-hastypes;
  collect-garbage;
  rho-eval-strict;
  debug;
  \Let(decls', result) -> result\
)
```

This is the strict evaluator, as explained in section 4.3.3.

```
rho-eval-strict = rec e(
  Let(id, rec x(is-nf; (not(is-con) <+
                rec y(Con(id) + App(y, x)))))
  + ( LetLet'(e)
//    + LetVar // non-sharing version
    + LetVar(e)
    + EvalLeft(e);
      (LeftChoice + LeftChoiceCut + RightChoice)
    + repeat(UncutRight);
      EvalArg(e);
      repeat(UncutLeft);
      EvalFunc(e);
      ( FailMatchP + FailMatchN
      <+ PropArg
      <+ Beta
      + Distrib
      + PropFunc
      + Let(id, is-con)
      + ConMatchP + ConMatchN
      + IntMatchP + IntMatchN
      + AppMatchP + AppMatchN
      )
    <+ \x -> <fatal-error> ["got stuck in term ", x]\
    ); e
  )
```

The remainder of this module is a garbage collector. It removes unused definitions from a let-expression. The standard strategy `graph-nodes` does all the hard work: calculating the transitive closure under the "the body of definition x uses y" relation, using the body of the expression as the root of the garbage collector.

Note that the garbage collector cannot be applied to just any let-expression — only the "top-level" let. For example, to evaluate **let** ds **in** $e_1 e_2$, we first evaluate **let** ds **in** e_1. But if we garbage-collect on the latter term, the variables appearing in e_2 will not be used as roots.

```
rules

  get-def: ("", Let(_, e)) -> e
  get-def: (x, Let(ds, _)) -> e
  where <LookupVar> (x, ds) => e

  used-vars: e -> <rho-free-vars; map(\Var(y) -> y\)> e

  add-def: ("", e, ds) -> ds
  add-def: (x, e, ds) -> [Def(x, e) | ds]

strategies

  collect-garbage =
    ?Let(ds, e);
    where (<printnl> (stderr, ["before gc: ", <length> ds]));
    where (<graph-nodes(get-def, used-vars, add-def)>
      ("", Let(ds, e), []) => ds');
    where (<printnl> (stderr, ["after gc:  ", <length> ds']));
    !Let(ds', e)
```

Appendix C

The RhoStratego type checker

The module `rho-typecheck` implements a type checker (or rather, a type inferencer) for RhoStratego. A number of improvements can be made to this implementation.

First, error messages could be improved vastly. Currently little information is given about the cause of the rejection of a program.

Second, the following program is rejected by the type checker:

```
x = id 123;
y = id "foo";
id = x -> x;
```

Placing `id` on top will make the program acceptable. Alternatively, the programmer could provide a type signature for `id`. The problem is that definitions are typed from top to bottom. When we type `x`, `id` is given the type `Int` $\rightarrow \alpha$, since we do not know that `id` is polymorphic. The application `id "foo"` will then be rejected.

The solution is to order the definitions in a list of *binding groups* (a smallest possible set of mutually recursive definitions), where no binding group refers to definitions in later binding groups [Jon99]. This will cause `id` to be typechecked (and given a polymorphic type) before `x` and `y`.

```
module rho-typecheck
imports lib rho rho-utils

strategies
```

We type-check a program by expanding type synonyms, putting all declarations in a let-expression with a dummy body, and then type-checking that let-expression. The initial environment Γ consists of the constructor type signatures. The variables in those type signatures must be renamed in order to prevent name clashes. The result of the type-checker is the original program with each term decorated with its type.

```
rho-typecheck = iowrap(
  ?Program(decls);
  where (<get-syns> decls => syns);
  where (<topdown(repeat(explode-syn(!syns)))> decls => decls');
  where (<get-tsigs; map(\(c, t) -> (Id(c, 0),
```

```
      <new-names> t)\)> decls' => initenv);
    ![initenv], Let(decls', Fail));
    rec i(infer(i) <+ \e -> <bad(!"unknown term")> e\);
    \(_, HasType(Let(decls'', _), _)) -> Program(decls'')\
  )
```

The strategy `explode-syn` expands type synonyms.

```
  explode-syn(syns) =
    where (syns => syns);
    listify-tapp; ?(TCon(c), targs);
    where (<lookup> (c, syns) => (vars, t));
    where (<zip(id)> (vars, targs) => subs);
    where (<subs> (subs, t) => t');
    !t'

rules
```

Environments are maintained as a *stack* of type assignments. The advantage is that we can easily add and remove groups of variables to and from the environment. The following strategies manage environments.

```
  push-env: (env, envs) -> [env | envs]
  pop-env: [env | envs] -> (env, envs)
  lookup-env: (nm, env) -> <lookup> (nm, <concat> env)
```

The strategy `infer` infers the type of a term, given a type environment. The code follows the algorithm given in section 6.3.4.
First, the trivial rules for typing literals.

```
  infer(i):
    (env, e@Const(Int(_))) -> (env, HasType(e, TCon("Int")))

  infer(i):
    (env, e@Const(Str(_))) -> (env, HasType(e, TCon("String")))
```

The Fail rule.

```
  infer(i):
    (env, e@Fail) -> (env, HasType(e, TVar(<new> ())))
```

The Con rule. Constructor types are carried around in the same environment as regular variables.

```
  infer(i):
    (env, e@Con(c)) -> (env, HasType(e, <findcon> (env, c)))
```

Primops have some random type α; this can be constrained using a type annotation or signature.

```
  infer(i):
    (env, e@PrimOp(s)) -> (env, HasType(e, TVar(<new> ())))
```

The CUT rule; just compute the type of the argument of the cut.

```
infer(i):
  (env, Cut(e)) -> (env', HasType(Cut(e'), t))
  where <i> (env, e) => (env', e'@HasType(_, t))
```

The VAR rule, following by an application of the INST rule, since all variables uses are instantiated automatically in the Hindley-Milner system.

```
infer(i):
  (env, e@Var(nm)) -> (env, HasType(e, <specialise> t))
  where
    <lookup-env <+ bad(!"undefined variable")> (nm, env) => t
```

The APP rule. If necessary, we apply the GEN rule to the type of the argument in order to support rank-2 types.

```
infer(i):
  (env, App(e1, e2)) ->
  (env'''''', HasType(App(e1', e2'), tres))
  where
    <i> (env, e1) => (env', e1'@HasType(_, t1));
    new => varg; new => vres;
    <push-env> ([("", TVar(varg)),
      ("", TVar(vres))], env') => env'';
    <supertype <+ bad(!"not a function")>
      (TFun(TVar(varg), TVar(vres)), t1) => s;
    <subs-env> (s, env'') => env''';
    <i> (env''', e2) => (env'''', e2'@HasType(_, t2));
    <Hd; Hd; Snd> env'''' => targ;
    <!(targ, t2); supertype
     <+ !(targ, <try-to-generalise> (env'''', t2)); supertype
     <+ bad(!"wrong argument")> () => s';
    <subs-env; pop-env> (s', env'''') =>
      ([(_, _), (_, tres)], env''''')
```

The CHOICE rule.

```
infer(i):
  (env, LChoice(e1, e2)) ->
  (env''''', HasType(LChoice(e1', e2'), t))
  where
    <i> (env, e1) => (env', e1'@HasType(_, t1));
    <push-env> ([("", t1)], env') => env'';
    <i> (env'', e2) => (env''', e2'@HasType(_, t2));
    <pop-env> env''' => ([("", t1')], env'''');
    <match-choice <+ Swap; match-choice <+
     bad(!"choice alts do not match")>
     (env'''', t1', t2) => (env''''', t)

match-choice: (env, t1, t2) -> (env', t)
  where
    <supertype> (t1, t2) => s;
    <subs> (s, t1) => t;
    <subs-env> (s, env) => env'
```

Type annotations.

```
infer(i):
  (env, HasType(e, t)) -> (env'', HasType(e', t''))
  where
    <i> (env, e) => (env', e'@HasType(_, t'));
    <supertype <+ bad(!"supertype in hastype")> (t, t') => s;
    <subs> (s, t) => t'';
    <subs-env> (s, env') => env''
```

The Abs rule.

```
infer(i):
  (env, Rule(pat, body)) ->
  (env''''', HasType(Rule(HasType(pat, tpat'), body'), t))
  where
    <rec p(pattype(p) <+ \p -> <bad(!"unknown pattern")> p\)>
      (env, pat) => (patenv, tpat);
    <push-env> (patenv, env) => env';
    <push-env> ([("", tpat)], env') => env'';
    <i> (env'', body) => (env''', body'@HasType(_, tbody));
    <pop-env> env''' => ([(_, tpat')], env'''');
    <pop-env> env'''' => (_, env''''');
    <contract> (env''''', TFun(tpat', tbody)) => t
```

The Let rule, with applications of the Gen rule to the inferred types of all definitions. The code is a bit complex because we have to deal with definitions without signatures as well as signatures without definitions (this is legal; the definitions may be given in different compilation units).

```
infer(i):
  (env, Let(decls, body)) ->
  (env, HasType(Let(decls'', body'), tbody))
  where
    <get-sigs> decls => localenv;
    <get-defs> decls => defs;

    <map(\(nm, _) -> <lookup; ![] <+
      ![(nm, TVar(<new> ()))]> (nm, localenv)
    \); concat> defs => localenv';

    <push-env> (<conc> (localenv, localenv'), env) => subenv;
    where (<foldl({t1, t1', t2, env, env', env'', e', s: \
      ((nm, e), (env, defs)) ->
      (env'', [Def(nm, e') | defs])
      where
        <i> (env, e) => (env', e'@HasType(_, t1));
        <debug(!"inferred: ")> t1;
        <try-to-generalise> (env', t1) => t1';
        <lookup-env> (nm, env) => t2;
        <supertype <+ bad(!"supertype in def")> (t2, t1') => s;
        <subs-env> (s, env') => env''
    \})> (defs, (subenv, [])) => (subenv', defs'));
    <i> (subenv', body) => (subenv'', body'@HasType(_, tbody));
```

```
    <pop-env> subenv'' => (localenv'', env');

    <filter(not(Def(id, id) + Sig(id, id)))> decls => decls';
    <map(\(nm, t) -> Sig(nm, t)\)> localenv'' => sigs;
    <concat> [decls', defs', sigs] => decls''
```

The strategy `pattype` computes the type of a pattern, as well as an environment containing type assignments for all variables occurring in the pattern. This code follows the typing rules in figure 6.3.

```
pattype(p): (_, Const(Int(_))) -> ([], TCon("Int"))

pattype(p): (_, Const(Str(_))) -> ([], TCon("String"))

pattype(p): (_, Fail) -> ([], TVar(<new> ()))

pattype(p): (_, Var(nm)) -> ([(nm, TVar(tvar))], TVar(tvar))
  where new => tvar

pattype(p): (inenv, p) -> (env, t')
  where
    <listify-app> p => (Con(c), ps);
    <findcon> (inenv, c) => t;
    <foldl(patapply(!inenv, p))> (ps, ([], t)) => (env, t');
    <not(?TFun(_, _)) <+
     bad(!"incomplete constructor pattern")> t'

patapply(inenv, p): (p, (env, t)) -> (env'', tres')
 where
   inenv => inenv;
   <?TFun(targ, tres) <+ bad(!"not a function pattern")> t;
   <p> (inenv, p) => (env', targ');
   <supertype <+ bad(!"bad pat argument")> (targ', targ) => s;
   <subs-env> (s, [<conc> (env, env')]) => [env''];
   <subs> (s, tres) => tres'

pattype(p):
  (inenv, p) ->
  ([(c, t0) | env], TGen(TVar(x0), <map(Snd)> env))
  where
    <listify-app> p => (Var(c), ps);
    <map(\Var(x) -> (x, TVar(<new> ()))\) <+
     bad(!"bad app pattern match")> ps => env;
    new => x0;
    <foldr(!TVar(x0), \((_, a), b) -> TFun(a, b)\)> env => t0

pattype(p):
  (inenv, HasType(pat, t)) -> (env', t'')
  where
    <p> (inenv, pat) => (env, t');
    <supertype <+ bad(!"bad hastype pat")> (t, t') => s;
    <subs> (s, t) => t'';
    <subs-env> (s, [env]) => [env']
```

```
pattype(p):
  (inenv, RTTC(t, pat)) -> (env, TRTTC(t))
  where <p> (inenv, pat) => (env, t')
```

The strategy `findcon` finds a constructor in the type environment and returns its type, with all type variables renamed.

```
findcon: (env, c) -> <new-names> t
  where
    <lookup-env <+
     bad(!"unknown constructor")> (Id(c, 0), env) => t
```

The strategy `specialise` implements the INST rule given in figure 6.2.

```
strategies

  specialise = outer-generalise; repeat(\TForAll(x, t) -> t\)
```

The strategy `try-to-generalise` implements (a sequence of applications of) the GEN rule. It generalises all free variables that do not occur in the environment.

```
rules

  try-to-generalise: (env, t) -> t'
    where
      <fvars; reverse> t => fv;
      <foldl(\(x, _t) -> <occurs-in-env; !_t <+ !TForAll(x, _t)>
        (env, x)\)> (fv, t) => t'
```

The strategy `supertype` implements the supertype function defined in section 6.3.4.

```
strategies

  supertype =
    (outer-generalise, outer-generalise); rec s(supertype(s))

rules

  supertype(s): (TCon(c), TCon(c)) -> []
  supertype(s): (TVar(x), TVar(x)) -> []
  supertype(s): (TVar(x), TVar(y)) -> [(x, TVar(y)), (y, TVar(x))]
  // x mag niet voorkomen in t!
  supertype(s): (TVar(x), t) -> [(x, t)]
  supertype(s): (t, TVar(x)) -> [(x, t)]
  supertype(s): (TApp(l1, r1), TApp(l2, r2)) -> <conc> (sl, sr)
    where
      <s> (l1, l2) => sl;
      <s> (<subs> (sl, r1), <subs> (sl, r2)) => sr
  supertype(s): (TFun(TVar(x), t2), TFun(TRTTC(t3), t4)) -> s'
    where
      <s> (TFun(TVar(x), t2), TFun(t3, t4)) => s;
      <filter((not(?x), id))> s => s'
  supertype(s): (TFun(t1, t2), TFun(TRTTC(t3), t4)) -> s
    where <s> (t2, t4) => s
```

```
  supertype(s): (TFun(t1, t2), TFun(t3, t4)) -> <conc> (s1, s2)
    where
      <s> (t3, t1) => s1;
      <s> (<subs> (s1, t2), <subs> (s1, t4)) => s2
  supertype(s): (TForAll(x1, t1), TForAll(x2, t2)) -> s
    where
      <s> (t1, <subs1> ((x2, TVar(x1)), t2)) => s;
      <not(lookup)> (x1, s)
  supertype(s): (t1, t2@TForAll(_, _)) -> s
    where
      <rho-trename> t2 => TForAll(x2', t2');
      <s> (t1, t2') => s
  supertype(s): (TForAll(x, t1), t2) -> s
    where <s> (t1, t2) => s; <not(lookup)> (x, s)
  supertype(s): t -> <fail> t
```

The strategies `subs` and `subs-env` perform substitutions over single types or entire type environments, respectively.

```
strategies

  subs = foldl(subs1)

  subs-env =
    ?(s, env); !env;
    map(map(\(nm, def) -> (nm, <subs> (s, def))\))

  subs1 = rec s(subs1(s))

rules

  subs1(s): ((x, t), TVar(x)) -> t
  subs1(s): ((x, t), TForAll(y, t2)) ->
            TForAll(y, <s> ((x, t), t2))
    where <not(eq)> (x, y)
  subs1(s): (sub, TApp(t1, t2)) ->
            TApp(<s> (sub, t1), <s> (sub, t2))
  subs1(s): (sub, TFun(t1, t2)) ->
            TFun(<s> (sub, t1), <s> (sub, t2))
  subs1(s): (sub, TRTTC(t)) -> TRTTC(<s> (sub, t))
  subs1(s): x@(sub, TGen(t, ts)) ->
            TGen(<s> (sub, t), <map(\t' -> <s> (sub, t')\)> ts)
  subs1(s): (_, t) -> t
```

The strategy `fvars` returns a list of free variables occurring in a type.

```
strategies

  fvars = rec f(fvars(f))

rules

  fvars(f): TVar(x) -> [x]
  fvars(f): TCon(_) -> []
```

```
  fvars(f): TApp(t1, t2) -> <union'> (<f> t1, <f> t2)
  fvars(f): TFun(t1, t2) -> <union'> (<f> t1, <f> t2)
  fvars(f): TRTTC(t) -> <f> t
  fvars(f): TForAll(x, t) -> <f; filter(not(?x))> t
  fvars(f): TGen(t, ts) -> <f> t
```

The strategy `occurs-in-env` succeeds if the specified variable occurs free in the environment.

```
strategies

  occurs-in-env =
    ?(env, x); !env; concat; map(Snd);
    fetch(fvars; fetch(?x))
```

The strategy `outer-generalise` floats quantifiers to the top of a type. For example, $A \to \forall\alpha.\alpha$ is transformed into the equivalent type $\forall\alpha.A \to \alpha$. This makes it easier to compare types.

```
strategies

  outer-generalise =
    rho-trename; rec x(bottomup(repeat(outer-generalise'(x))))

rules

  outer-generalise'(x):
    TFun(t1, TForAll(x, t2)) -> TForAll(x, <x> TFun(t1, t2))
```

The strategy `contract` implements the CONTRACT rule.

```
strategies

  contract = ?(env, t); !t; topdown(try(contract'(!env, !t)))

rules

  contract'(env, t'): TGen(t, ts) -> t
    where
      env => env; t' => t';
      <map({x: ?TVar(x); where
        (<not(occurs-in-env)> (env, x))})> [t | ts];
      <map({x: ?TVar(x); where
        (<not(occurs-in-env)> ([[("", t')]], x))})> ts
```

The remainder is support code.

```
rules

  listify-tapp: t -> (t0, <reverse> ts)
    where
      <rec x(\TApp(a, b) -> [b | <x> a]\ <+ ?t0; ![])> t => ts

  listify-app: e -> (e0, <reverse> es)
```

```
    where
      <rec x(\App(a, b) -> [b | <x> a]\ <+ ?e0; ![])> e => es

  union': (l1, l2) -> <conc> (l1,
            <filter(not({x: ?x; !l1; fetch(?x)}))> l2)

  new-names: t -> <try-to-generalise; specialise> ([], t)

strategies

  bad(msg) = debug; !["type error: ", <msg> ()]; fatal-error
```

Appendix D

The RhoStratego standard library

The file `stdlib.rho` contains the implementation of the RhoStratego standard library. Type signatures and data types are defined in the header file `stdlib.rh`.

```
#include "stdlib.rh"
```

Simple functions

```
id = x -> x;

const = x -> y -> x;

if = (True -> e1 -> e2 -> ^e1) <+
     (False -> e1 -> e2 -> e2);

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

List operations

```
foldr = op -> nul ->
  ([] -> nul <+
   x : xs -> op x (foldr op nul xs));

map = f ->
  foldr (x -> xs -> (f x) : xs) [];

filter = f -> foldr (x -> xs -> if (f x) (x : xs) xs) [];

elem = x -> foldr (y -> b -> (x == y) || b) False;

union = xs -> ys -> (
  let f = z -> zs -> if (z `elem` zs) zs (z : zs);
  in foldr f ys xs);

conc = xs -> ys -> foldr (x -> y -> x : y) ys xs;
```

```
concat = xss -> foldr conc [] xss;

unions = xss -> foldr union [] xss;

diff = xs -> ys -> filter (x -> not (x `elem` ys)) xs;

lookup = x -> foldr (<x', e> -> e' -> if (x == x') e e') fail;
```

Strict application

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

Strategy operators

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

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

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

Hyperstrictness

```
force = all force;
```

Traversal primitives

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

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

Traversal

```
topdown = (s :: (a . a -> a)) -> s | all (topdown s);

bottomup = (s :: (a . a -> a)) -> all (bottomup s) | s;

oncetd = (s :: (a . a -> a)) -> (s <+ one (oncetd s));

mapkids = (s :: (c . c -> a)) ->
  (c x -> (s x) : (mapkids s c) <+ x -> []);

crush = op -> nul -> (s :: (a . a -> b)) -> x ->
  foldr op nul (mapkids s x);

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

Tuples

```
fst = <a, b> -> a;
```

```
snd = <a, b> -> b;
```

Booleans

```
not = True -> False <+ False -> True;
&& = True -> True -> True <+ x -> y -> False;
|| = False -> False -> False <+ x -> y -> True;

b2f = True -> True <+ x -> fail;
f2b = fail -> False <+ x -> True;
```

Equality

The function == defines equality generically over all terms.

```
== = c x -> d y -> ^((c == d) && (x == y))
     <+ x -> y -> primOp "p_primeq";
!= = x -> y -> not (x == y);
```

Arithmetic

```
+ = x -> y -> primOp "p_add";
- = x -> y -> primOp "p_sub";
* = x -> y -> primOp "p_mul";
/ = x -> y -> primOp "p_div";
> = x -> y -> primOp "p_gt";
< = x -> y -> (y > x);
```

Monadic I/O

```
stdin = 0;
stdout = 1;
stderr = 2;

hPutStr = handle -> s -> w -> primOp "p_putstr";

hPrint = handle -> term -> w -> primOp "p_print";

putStr = hPutStr stdout;

print = hPrint stdout;

readFile = filename -> w -> primOp "p_readfile";

writeFile = filename -> s -> w -> primOp "p_writefile";


>>= = m1 -> (m2 :: (a -> IO b)) -> w ->
  (<w, x> -> m2 x w) (m1 w);

>> = m1 -> (m2 :: IO b) -> m1 >>= (_ -> m2);
```

```
return = x -> w -> <w, x>;

unsafePerformIO = m -> (<w, x> -> x) (m World);

debug = x -> unsafePerformIO (hPrint stderr x >> return x);
```

ATerm interface

```
readTerm = filename -> w -> primOp "p_readterm" :: <World, a>;

writeTerm = filename -> fmt -> term -> w ->
  primOp "p_writeterm";
```

Bibliography

[AS96] Andrew W. Appel and Zhong Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, 1996.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bar84] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume II of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, second edition, 1984.

[BMS$^+$] Warren Burton, Erik Meijer, Patrick Sansom, Simon Thompson, and Philip Wadler. Views: An extension to Haskell pattern matching. `http://www.haskell.org/development/views.html`.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[vdBdJKO00] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software—Practice and Experience*, 2000.

[Cir00] Horatio Cirstea. *Calcul de réécriture: fondements et applications*. PhD thesis, Université Henri Poincaré – Nancy 1, October 2000.

[CK98] Horatiu Cirstea and Claude Kirchner. ρ-Calculus: Its syntax and basic properties. Technical Report 98-R-218, LORIA, Nancy (France), August 1998.

[CKL01] Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Matching power. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*. Springer-Verlag, May 2001.

[dJVV01] Merijn de Jonge, Eelco Visser, and Joost 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/`.

[EP00] Martin Erwig and Simon Peyton Jones. Pattern guards and transformational patterns. In *Haskell Workshop*, 2000.

[HP00] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Haskell Workshop*, Montreal, Canada, September 2000.

[Jon99] Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, September 1999. Latest version (September 2000): `http://www.cse.ogi.edu/~mpj/thih/`.

[Läm01] Ralf Lämmel. Generic type-preserving traversal strategies. In Bernhard Gramlich and Salvador 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.

[LV00] Ralf Lämmel and Joost Visser. Type-safe functional strategies. In *Scottish Functional Programming Workshop*, July 2000.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Pey92] Simon Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[PH+99] Simon Peyton Jones, John Hughes, et al. Report on the programming language Haskell 98, 1999.

[PRHM99] Simon Peyton Jones, Alastair Reid, Tony Hoare, and Simon Marlow. A semantics for imprecise exceptions. In *ACM Conference on Programming Languages Design and Implementation*, pages 25–36, 1999.

[PRR99] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, October 1999.

[PS98] Simon Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, September 1998.

[PW93] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *20th ACM Symp. on Principles of Programming Languages (POPL'93)*, pages 71–84, Charlotte, North Carolina, January 1993.

[RP00] Norman Ramsey and Simon 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.

[Ste78] G.L. Steele. RABBIT: A compiler for SCHEME. Technical Report AI-TR-474, MIT AI Lab., May 1978.

[Tul00] Mark Tullsen. First class patterns. In *2nd International Workshop on Practial Aspects of Declarative Languages*, volume 1753 of *LNCS*, pages 1–15, 2000.

[VeABT98] Eelco Visser, Zine el Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming (ICFP'98)*, ACM SIGPLAN, pages 13–26, September 1998.

[Vis] Eelco Visser. Stratego language website. `http://www.stratego-language.org`.

[Vis97] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[Vis00] Eelco Visser. Language independent traversals for program transformation. In Johan Jeuring, editor, *Workshop on Generic Programming (WGP2000)*, July 2000.

[Vis01a] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. system description for Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*. Springer-Verlag, May 2001.

[Vis01b] Eelco Visser. A survey of strategies in program transformation systems. In B. Gramlich and S. Lucas Alba, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, May 2001.

[Wad87] Philip 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.

[Wad92] Philip Wadler. The essence of functional programming (invited talk). In *19th ACM Symp. on Principles of Programming Languages (POPL'92)*, Albuquerque, New Mexico, January 1992.

[Wel99] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.