Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Typed combinators for generic traversal

R. Lämmel, J.M.W. Visser

REPORT SEN-R0124 AUGUST 2001

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Typed Combinators for Generic Traversal

Ralf Lämmel and Joost Visser
Email: (Ralf.Laemmel|Joost.Visser)@cwi.nl
WWW: http://www.cwi.nl/~(ralf|jvisser)/

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

Lacking support for generic traversal, functional programming languages suffer from a scalability problem when applied to large-scale program transformation problems. As a solution, we introduce *functional strategies*: typeful generic functions that not only can be applied to terms of any type, but which also allow generic traversal into subterms. We show how strategies are modelled inside a functional language, and we present a combinator library including generic traversal combinators. We illustrate our technique of programming with functional strategies by an implementation of the *extract method* refactoring for Java.

1998 ACM Computing Classification System: D.1.1, D.1.2, D.2.3, D.2.13, D.3.3, I.1.1, I.1.2, I.1.3, I.2.2
Keywords and Phrases: Genericity, traversal, combinators, program transformation

1. Introduction

Our domain of interest is program transformation in the context of software re-engineering [5, 1, 4]. Particular problems include automated refactoring (e.g., removal of duplicated code, or goto elimination) and conversion (e.g., Cobol 74 to 85, or Euro conversion). In this context, the bulk of the functionality consists of traversal over the syntax of the involved languages. Most problems call for various different traversal schemes. The involved syntaxes are typically complex (50-2000 grammar productions), and often one has to cope with evolving languages, diverging dialects and embedded languages. In such a setting, genericity regarding traversal is indispensable [4, 18].

By lack of support for generic term traversal, functional programming suffers from a serious and notoriously ignored scalability problem when applied to program transformation problems. To remedy this situation, we introduce functional *strategies*: generic functions that cannot only (i) be applied to terms of any type, but which also (ii) allow generic traversal into subterms, and (iii) may exhibit non-generic (ad-hoc) behaviour for particular types. We show how these strategies can be modelled inside the functional language Haskell[1], and we present a strategy combinator library that includes traversal combinators.

A generic traversal problem Let us consider a simple traversal problem and its solution. Assume we want to accumulate all the variables on use sites in a given abstract syntax tree of a Java program. We envision a traversal which is independent of the Java syntax except that it must be able to identify Java variables on use sites. Here is a little Java fragment:

```
//print details
System.out.println("name:" + _name);
System.out.println("amount" + amount);
```

[1] Throughout the paper we use Haskell 98 [11] extended with rank-2 polymorphism as available in Hugs and GHC.

The traversal should return the list `["_name","amount"]` of variables on use sites. Such collection of variables can be based on a simple and completely generic traversal scheme of the following name and type:

$$collect :: MonadPlus\ m \Rightarrow TU\ m\ [a] \rightarrow TU\ m\ [a]$$

Here, $TU\ m\ [a]$ is the type of *type-unifying* generic functions which map terms of any type to a list of as. Besides type-unifying strategies like the above *collect*, we will later encounter so-called *type-preserving* strategies where input and output type coincide. The strategy combinator *collect* maps a type-unifying strategy intended for identification of collectable entities to a type-unifying strategy performing the actual collection. We use the combinator in the following manner to collect Java variables on use sites:

$$\begin{array}{lcl} collectUseVars & :: & TU\ Maybe\ [String] \\ collectUseVars & = & collect\ (monoTU\ useVar) \\ useVar & :: & Expression \rightarrow Maybe\ [String] \\ useVar\ (Identifier\ i) & = & Just\ [i] \\ useVar\ \_ & = & Nothing \end{array}$$

The traversal *collectUseVars* can be applied to any kind of Java program fragment, and it will return the variables identified by *useVar*. Note that the combinator *collect* is not applied directly to the function *useVar*. We first need to extend this monomorphic function to be applicable to terms of all sorts. This extension is performed by the combinator *monoTU*.

Generic functional programming Note that the code above does not mention any of Java's syntactical constructs except the syntax of identifiers relevant to the problem. Traversal over the other constructs is accomplished with the fully generic traversal scheme *collect*. As a consequence of this genericity, the solution to our example program is extremely concise and declarative. In the sequel, we will demonstrate how generic combinators like *collect* are defined and how they are used to construct generic functional programs that solve non-trivial program transformation problems.

Structure of the article In Section 2 we model strategies with abstract data types (ADTs) to be implemented later, and we explain the primitive and defined strategy combinators offered by our strategy library. In Section 3, we illustrate the utility of generic traversal combinators for actual programming by an implementation of an automated program refactoring. In Section 4, we study two implementations for the strategy ADTs, namely an implementation based on a hidden universal term representation, and an implementation that relies on a combination of first-class polymorphism and an encoding technique for type cases. The paper is concluded in Section 5. The complete code of the strategy library and some further illustrative code is included in Appendices I and II.

Acknowledgements We are grateful to Johan Jeuring for discussions on the subject.

2. A STRATEGY LIBRARY

We present a library for generic programming with strategies. To this end, we introduce ADTs with primitive combinators for strategies (i.e., generic functions). For the moment, we consider the representation of strategies as opaque since different models are possible as we will see in Section 4. The primitive combinators cover concepts we are used to for ordinary functions, namely application and sequential composition. There are further important facets of strategies, namely partiality or non-determinism, and access to the immediate subterms of a given term. Especially the latter facet makes clear that strategies go beyond parametric polymorphism. A complete overview of all primitive strategy combinators is shown in Figure 1. After a discussion of the primitives, we will discuss a number of defined strategies, especially traversal schemes.

Strategy types (opaque)

$$\begin{array}{ll} \textbf{data}\ \mathit{Monad}\ m & \Rightarrow\ \mathit{TP}\ m = \ldots\ \mathit{abstract} \\ \textbf{data}\ \mathit{Monad}\ m & \Rightarrow\ \mathit{TU}\ m\ a = \ldots\ \mathit{abstract} \end{array}$$

Strategy application

$$\begin{array}{ll} \mathit{applyTP} :: (\mathit{Monad}\ m, \mathit{Term}\ t) & \Rightarrow\ \mathit{TP}\ m \to t \to m\ t \\ \mathit{applyTU} :: (\mathit{Monad}\ m, \mathit{Term}\ t) & \Rightarrow\ \mathit{TU}\ m\ a \to t \to m\ a \end{array}$$

Strategy construction

$$\begin{array}{ll} \mathit{polyTP} :: \mathit{Monad}\ m & \Rightarrow\ (\forall x.\ x \to m\ x) \to \mathit{TP}\ m \\ \mathit{polyTU} :: \mathit{Monad}\ m & \Rightarrow\ (\forall x.\ x \to m\ a) \to \mathit{TU}\ m\ a \\ \mathit{adhocTP} :: (\mathit{Monad}\ m, \mathit{Term}\ t) & \Rightarrow\ \mathit{TP}\ m \to (t \to m\ t) \to \mathit{TP}\ m \\ \mathit{adhocTU} :: (\mathit{Monad}\ m, \mathit{Term}\ t) & \Rightarrow\ \mathit{TU}\ m\ a \to (t \to m\ a) \to \mathit{TU}\ m\ a \end{array}$$

Sequential composition

$$\begin{array}{ll} \mathit{seqTP} :: \mathit{Monad}\ m & \Rightarrow\ \mathit{TP}\ m \to \mathit{TP}\ m \to \mathit{TP}\ m \\ \mathit{letTP} :: \mathit{Monad}\ m & \Rightarrow\ \mathit{TU}\ m\ a \to (a \to \mathit{TP}\ m) \to \mathit{TP}\ m \\ \mathit{seqTU} :: \mathit{Monad}\ m & \Rightarrow\ \mathit{TP}\ m \to \mathit{TU}\ m\ a \to \mathit{TU}\ m\ a \\ \mathit{letTU} :: \mathit{Monad}\ m & \Rightarrow\ \mathit{TU}\ m\ a \to (a \to \mathit{TU}\ m\ b) \to \mathit{TU}\ m\ b \end{array}$$

Choice

$$\begin{array}{ll} \mathit{choiceTP} :: \mathit{MonadPlus}\ m & \Rightarrow\ \mathit{TP}\ m \to \mathit{TP}\ m \to \mathit{TP}\ m \\ \mathit{choiceTU} :: \mathit{MonadPlus}\ m & \Rightarrow\ \mathit{TU}\ m\ a \to \mathit{TU}\ m\ a \to \mathit{TU}\ m\ a \end{array}$$

Traversal combinators

$$\begin{array}{ll} \mathit{allTP} :: \mathit{Monad}\ m & \Rightarrow\ \mathit{TP}\ m \to \mathit{TP}\ m \\ \mathit{oneTP} :: \mathit{MonadPlus}\ m & \Rightarrow\ \mathit{TP}\ m \to \mathit{TP}\ m \\ \mathit{allTU} :: (\mathit{Monad}\ m, \mathit{Monoid}\ a) & \Rightarrow\ \mathit{TU}\ m\ a \to \mathit{TU}\ m\ a \\ \mathit{oneTU} :: \mathit{MonadPlus}\ m & \Rightarrow\ \mathit{TU}\ m\ a \to \mathit{TU}\ m\ a \end{array}$$

Figure 1: Primitive strategy combinators.

2.1 Strategy types and application

There are two kinds of strategies. Firstly, the ADT $\mathit{TP}\ m$ models type-preserving strategies where the result of a strategy application to a term of type t is of type $m\ t$. Secondly, the ADT $\mathit{TU}\ m\ a$ models type-unifying strategies where the result of strategy application is always of type $m\ a$ regardless of the type of the input term. These contracts are expressed by the types of the corresponding combinators *applyTP* and *applyTU* for strategy application (*cf.* Figure 1). In both cases, m is a monad parameter [27] to deal with effects in strategies such as state passing or non-determinism. Also note that we do not apply strategies to arbitrary types but only to instances of the class *Term* for term types. This is sensible since we ultimately want to traverse into subterms. Recall that the introductory example is a type-unifying traversal with the result type $[\mathit{String}]$.

2.2 Strategy construction

There are two ways to construct strategies from ordinary functions. Firstly, one can turn a parametric polymorphic function into a strategy (*cf.* *polyTP* and *polyTU* in Figure 1). Secondly, one can *update* a strategy to apply a monomorphic function for a given type to achieve type-dependent behaviour (*cf.* *adhocTP* and *adhocTU*). In other words, one can dynamically provide ad-hoc cases for a strategy.

Let us first illustrate the construction of strategies from parametric polymorphic functions:

$$\begin{array}{lcl} \mathit{identity} & :: & \mathit{Monad}\ m \Rightarrow \mathit{TP}\ m \\ \mathit{identity} & = & \mathit{polyTP}\ \mathit{return} \end{array} \qquad \begin{array}{lcl} \mathit{build} & :: & \mathit{Monad}\ m \Rightarrow a \to \mathit{TU}\ m\ a \\ \mathit{build}\ a & = & \mathit{polyTU}\ (\mathit{const}\ (\mathit{return}\ a)) \end{array}$$

The type-preserving strategy *identity* denotes the generic (and monadic) identity function. The type-unifying strategy *build a* denotes the generic function which returns a regardless of the input term. As a consequence of parametricity [26], there are no further ways to inhabit the argument types of *polyTP* and *polyTU* (unless we rely on a specific instance of m).

The second way of strategy construction, i.e., with the *adhoc* combinators, allows us to go beyond parametric polymorphism. Given a strategy, we can provide an ad-hoc case for a specific type. Here is a simple example:

$$\begin{array}{lcl} \mathit{gnot} & :: & \mathit{Monad}\ m \Rightarrow \mathit{TP}\ m \\ \mathit{gnot} & = & \mathit{adhocTP}\ \mathit{identity}\ (\mathit{return} \circ \mathit{not}) \end{array}$$

The strategy *gnot* is applicable to terms of any type. It will behave like *identity* most of the time, but it will perform Boolean negation when faced with a Boolean. Such type cases are crucial to assemble traversal strategies that exhibit specific behaviour for certain types of the traversed syntax.

2.3 Sequential composition

Since the strategy types are opaque, sequential composition has to be defined as a primitive concept. This is in contrast to ordinary functions where one can define function composition in terms of λ-abstraction and function application. Consider the following parametric polymorphic forms of sequential composition:

$$\begin{array}{lcl} g \circ f & = & \lambda x \to g\ (f\ x) \\ f\ `\mathit{mseq}`\ g & = & \lambda x \to f\ x \gg\!= g \\ f\ `\mathit{mlet}`\ g & = & \lambda x \to f\ x \gg\!= \lambda y \to g\ y\ x \end{array}$$

The first form describes ordinary function composition. The second form describes the monadic variation. The third form can be regarded as a let-expression with a free variable x. An input for x is passed to both f and g, and the result of the first application is fed to the second function. The latter two polymorphic forms of sequential composition serve as prototypes of the strategic combinators for sequential composition. The strategy combinators *seqTP* and *seqTU* of Figure 1 correspond to *mseq* lifted to the strategy level. Note that the first strategy is always a type-preserving strategy. The strategy combinators *letTP* and *letTU* are obtained by lifting *mlet*. Note that the first strategy is always a type-unifying strategy. The necessary kind of lifting will be explained when we discuss possible implementations of the strategy types.

Let us illustrate the utility of *letTU*. We want to lift a binary operator o to the level of type-unifying strategies by applying two argument strategies to the same input term and combining their intermediate results by o. Here is the corresponding strategy combinator:

$$\begin{array}{lcl} \mathit{comb} & :: & \mathit{Monad}\ m \Rightarrow (a \to b \to c) \to \mathit{TU}\ m\ a \to \mathit{TU}\ m\ b \to \mathit{TU}\ m\ c \\ \mathit{comb}\ o\ s\ s' & = & s\ `\mathit{letTU}`\ \lambda a \to s'\ `\mathit{letTU}`\ \lambda b \to \mathit{build}\ (o\ a\ b) \end{array}$$

2.4 Partiality and non-determinism

Instead of the simple class *Monad* we can also consider strategies w.r.t. the extended class *MonadPlus* with the members *mplus* and *mzero*. This provides us with means to express partiality and non-determinism. It is often useful to consider strategies which might potentially fail. The following ordinary function combinator is the prototype for the *choice* combinators in Figure 1.

$$f\ `mchoice`\ g \quad = \quad \lambda x \to (f\ x)\ `mplus`\ (g\ x)$$

As an illustration let us define three simple strategy combinators which contribute to the construction of the introductory example.

$$\begin{array}{lcl} failTU & :: & MonadPlus\ m \Rightarrow TU\ m\ x \\ failTU & = & polyTU\ (const\ mzero) \\ monoTU & :: & (Term\ a, MonadPlus\ m) \Rightarrow (t \to m\ a) \to TU\ m\ a \\ monoTU\ f & = & adhocTU\ failTU\ f \\ tryTU & :: & (MonadPlus\ m, Monoid\ a) \Rightarrow TU\ m\ a \to TU\ m\ a \\ tryTU\ s & = & s\ `choiceTU`\ (build\ mempty) \end{array}$$

The strategy *failTU* denotes unconditional failure. The combinator *monoTU* updates failure by a monomorphic function f, using the combinator *adhocTU*. That is, the resulting strategy fails for all types other than f's argument type. If f is applicable, then the strategy indeed resorts to f. The combinator *tryTU* allows us to recover from failure in case we can employ a neutral element *mempty* of a monoid.

2.5 Traversal combinators

A challenging facet of strategies is that they might traverse terms. In fact, any program transformation or program analysis involves traversal. If we want to employ genericity for traversal, corresponding basic combinators are indispensable. The *all* and *one* combinators in Figure 1 process all or just one of the *immediate* subterms of a given term, respectively. The combinators do not just vary with respect to quantification but also for the type-preserving and the type-unifying case. The type-preserving combinators *allTP* and *oneTP* preserve the outermost constructor for the sake of type-preservation. Dually, the type-unifying combinators *allTU* and *oneTU* unwrap the outermost constructor in order to migrate to the unified type. More precisely, *allTU* reduces all pre-processed children by the binary operation *mappend* of a monoid whereas *oneTU* returns the result of processing one child. The *all* and *one* combinators have been adopted from the untyped language Stratego [25] for strategic term rewriting.

We are now in the position to define the traversal scheme *collect* from the introduction. We first define a more parametric strategy *crush* which performs a deep reduction by employing the operators of a monoid parameter. Then, the strategy *collect* is nothing more than a type-specialized version of *crush* where we opt for the list monoid.

$$\begin{array}{lcl} crush & :: & (MonadPlus\ m, Monoid\ a) \Rightarrow TU\ m\ a \to TU\ m\ a \\ crush\ s & = & comb\ mappend\ (tryTU\ s)\ (allTU\ (crush\ s)) \\ collect & :: & MonadPlus\ m \Rightarrow TU\ m\ [a] \to TU\ m\ [a] \\ collect\ s & = & crush\ s \end{array}$$

2.6 Some defined combinators

We can subdivide defined combinators into two categories, one for the control of strategies, and another for traversal schemes. Let us discuss a few examples of defined combinators. A complete library can be found in Appendix I.3. Here are some representatives of the category for the control of strategies:

$$\begin{array}{lcl} repeatTP & :: & MonadPlus\ m \Rightarrow TP\ m \to TP\ m \\ repeatTP\ s & = & tryTP\ (seqTP\ s\ (repeatTP\ s)) \\ ifthenTP & :: & Monad\ m \Rightarrow TP\ m \to TP\ m \to TP\ m \\ ifthenTP\ f\ g & = & (f\ `seqTU`\ (build\ ()))\ `letTP`\ (const\ g) \end{array}$$

$$
\begin{array}{lcl}
notTP & :: & MonadPlus\ m \Rightarrow TP\ m \to TP\ mentity \\
notTP\ s & = & ((s\ `ifthenTU`\ (build\ True))\ `choiceTU`\ (build\ False)) \\
 & & `letTP`\lambda b \to \textbf{if}\ b\ \textbf{then}\ failTP\ \textbf{else}\ identity \\
afterTU & :: & Monad\ m \Rightarrow (a \to b) \to TU\ m\ a \to TU\ m\ b \\
afterTU\ f\ s & = & s\ `letTU`\ \lambda a \to build\ (f\ a)
\end{array}
$$

The combinator *repeatTP* applies its argument strategy as often as possible. As an aside, a type-unifying counter-part of this combinator would justly not be typeable. The combinator *ifthenTP* precedes the application of a strategy by a guarding strategy. The guard determines whether the guarded strategy is applied at all. However, the guarded strategy is applied to the original term (as opposed to the result of the guarding strategy). The combinator *notTP* models negation by failure. The combinator *afterTU* adapts the result of a type-unifying traversal by an ordinary function.

Let us also define a few traversal schemes (in addition to *crush* and *collect*):

$$
\begin{array}{lcl}
bu & :: & Monad\ m \Rightarrow TP\ m \to TP\ m \\
bu\ s & = & (allTP\ (bu\ s))\ `seqTP`\ s \\
oncetd & :: & MonadPlus\ m \Rightarrow TP\ m \to TP\ m \\
oncetd\ s & = & s\ `choiceTP`\ (oneTP\ (oncetd\ s)) \\
select & :: & MonadPlus\ m \Rightarrow TU\ m\ a \to TU\ m\ a \\
select\ s & = & s\ `choiceTU`\ (oneTU\ (select\ s)) \\
selectenv & :: & MonadPlus\ m \Rightarrow e \to (e \to TU\ m\ e) \to (e \to TU\ m\ a) \to TU\ m\ a \\
selectenv\ e\ s'\ s & = & s'\ e\ `letTU`\ \lambda e' \to (s\ e)\ `choiceTU`\ (oneTU\ (selectenv\ e'\ s'\ s))
\end{array}
$$

All these schemes deal with recursive traversal. The combinator *bu* serves for unconstrained type-preserving bottom-up traversal. The argument strategy has to succeed for every node if the traversal should succeed. The combinator *oncetd* serves for type-preserving top-down traversal where the argument strategy is tried until it succeeds once. The traversal fails if the argument strategy fails for all nodes. The type-unifying combinator *select* searches in top-down manner for a node which can be processed by the argument strategy. Finally, the combinator *selectenv* is an elaboration of *select* to accomplish explicit environment passing. The first argument strategy serves for updating the environment before descending into the subterms.

3. Application: Refactoring

Refactoring [10] is the process of step-wise improving the internal structure of a software system without altering its external behaviour. The *extract method refactoring* [10, p. 110] is a well-known example of a basic refactoring step. To demonstrate the technique of programming with strategy combinators, we will implement the extract method refactoring for Java.

3.1 The extract method *refactoring*

In brief, the extract method refactoring is described as follows:

> *Turn a code fragment that can be grouped together into a reusable method whose name explains the purpose of the method.*

For instance, the last two statements in the following method can be grouped into a method called `printDetails`.

```
void printOwning(double amount) {
  printBanner ();
  //print details
  System.out.println("name:" + _name);
  System.out.println("ammount" + amount);
}
```

```
void printOwning(double amount) {
  printBanner ();
  printDetails(amount);
}
void printDetails(double amount) {
  System.out.println("name:" + _name);
  System.out.println("amount" + amount);
}
```

Note that the local variable `amount` is turned into a parameter of the new method, while the instance variable `_name` is not. Note also, that the *extract method* refactoring is valid only for a code fragment that does not contain any return statements or assignments to local variables.

3.2 Design of the algorithm

To implement the *extract method* refactoring, we need to solve a number of subtasks.

Legality check The focused fragment must be analysed to ascertain that it does not contain any return statements or assignments to local variables. The latter involves detection of variables in the fragment that are defined (assigned into), but not declared (i.e., free *defined* variables).

Generation The new method declaration and invocation need to be generated. To construct their formal and actual parameter lists, we need to collect those variables that are used, but not declared (i.e., free *used* variables) from the focused fragments, with their types.

Transformation The focused fragment must be replaced with the generated method invocation, and the generated method declaration must be inserted in the class body.

These subtasks need to be performed at specific moments during a traversal of the abstract syntax tree. Roughly, our traversal will be structured as follows:

1. Descend to the class declaration in which the method with the focused fragment occurs.
2. Descend into the method with the focused fragment to (i) check the legality of the focused fragment, and (ii) return both the focused fragment and a list of typed free variables that occur in the focus.
3. Descend again to the focus to replace it with the method invocation that can now be constructed from the list of typed free variables.

3.3 Implementation with strategies

Our solution is shown in Figures 2 through 4.

```
typed_free_vars :: (MonadPlus m, Eq v)
                 ⇒ [(v, t)] → TU m [v] → TU m [(v, t)] → TU m [(v, t)]
typed_free_vars env getvars declvars
  = afterTU (flip appendMap env) (tryTU declvars) `letTU` λenv' →
      choiceTU (afterTU (flip selectMap env') getvars)
               (comb diffMap (allTU (typed_free_vars env' getvars declvars))
                             (tryTU declvars))
```

Figure 2: A generic algorithm for extraction of free variables with their declared types.

```
useVar (Identifier i)     = return [i]
useVar _                  = mzero
defVar (Assignment i _)   = return [i]
declVars                  :: MonadPlus m ⇒ TU m [(Identifier, Type)]
declVars                  = adhocTU (monoTU declVarsBlock) declVarsMeth
                            where declVarsBlock (BlockStatements vds _)
                                    = return (map (λ(VariableDecl t i) → (i, t)) vds)
                                  declVarsMeth (MethodDecl _ _ (FormalParams fps) _)
                                    = return (map (λ(FormalParam t i) → (i, t)) fps)
freeUseVars env           = afterTU nubMap (typed_free_vars env (monoTU useVar) declVars)
freeDefVars env           = afterTU nubMap (typed_free_vars env (monoTU defVar) declVars)
```

Figure 3: Instantiations of the generic free variable algorithm for Java.

Free variable analysis As noted above, we need to perform two kinds of free variable collection: variables used but not declared, and variables defined but not declared. Furthermore, we need to find the types of these free variables. Using strategies, we can implement free variable collection in an extremely generic fashion. Figure 2 shows a generic free variable collection algorithm. This algorithm was adapted from an untyped rewriting strategy in [24]. It is parameterized with (i) an initial type environment *env*, (ii) a strategy *getvars* which selects any variables that are used in a certain node of the AST, and (iii) a strategy *declvars* which selects declared variables with their types. Note that no assumptions are made with respect to variables or types, except that equality is defined on variables so they can appear as keys in a map.

The algorithm basically performs a top-down traversal. At a given node, first the incoming type environment is extended with any variables declared at this node. Second, either the variables used at the node are looked-up in the type environment and returned with their types, or, if the node is not a use site, any declared variables are subtracted from the collection of free variables found in the children. Note that the algorithm is typeful, and fully generic. It makes ample use of library combinators, such as *afterTU*, *letTU* and *comb*.

As shown in Figure 3, this generic algorithm can be instantiated to the two kinds of free variable analysis needed for our case. The functions *useVar*, *defVar*, and *declVars* are the Java-specific ingredients that are needed. They determine the used, defined, and declared variables of a given node, respectively. We use them to instantiate the generic free variable collector to construct *freeUseVars*, and *freeDefVars*.

```
extractMethod :: (Term t, MonadPlus m) ⇒ t → m t
extractMethod prog = applyTP (oncetd (monoTP extrMethFromCls)) prog

extrMethFromCls :: MonadPlus m ⇒ ClassDeclaration → m ClassDeclaration
extrMethFromCls (ClassDecl fin nm sup fs cs ds)
  = do (pars, body) ← ifLegalGetParsAndBody ds
       ds' ← replaceFocus pars (ds ++ [constructMethod pars body])
       return (ClassDecl fin nm sup fs cs ds')

ifLegalGetParsAndBody :: (Term t, MonadPlus m) ⇒ t → m ([([Char], Type)], Statement)
ifLegalGetParsAndBody ds
  = applyTU (selectenv [] appendLocals ifLegalGetParsAndBody1) ds
    where ifLegalGetParsAndBody1 env
            = getFocus `letTU` λs →
              ifthenTU (isLegal env)
              (freeUseVars env `letTU` λpars →
               build (pars, s))
          appendLocals env
            = comb appendMap (tryTU declVars) (build env)

replaceFocus :: (Term t, MonadPlus m) ⇒ [(Identifier, Type)] → t → m t
replaceFocus pars ds = applyTP (oncetd (replaceFocus1 pars)) ds
  where replaceFocus1 pars
          = getFocus `letTP` λ_ →
            monoTP (const (return (constructMethodCall pars)))

isLegal :: MonadPlus m ⇒ [([Char], Type)] → TP m
isLegal env = freeDefVars env `letTP` λenv' →
              if null env' then notTU (select getReturn) else failTP

getFocus :: MonadPlus m ⇒ TU m Statement
getFocus = monoTU (λs → case s of (StatFocus s') → return s'
                                  _ → mzero)

getReturn :: MonadPlus m ⇒ TU m (Maybe Expression)
getReturn = monoTU (λs → case s of (ReturnStat x) → return x
                                   _ → mzero)
```

Figure 4: Implementation of the *extract method* refactoring.

Method extraction The remainder of the extract method implementation is shown in Figure 4. The main strategy *extractMethod* performs a top-down traversal to the class level, where it calls *extrMethFromCls*. This latter function first obtains parameters and body with *ifLegalGetParsAndBody*, and then replaces the focus with *replaceFocus*. Code generation is performed by two functions *constructMethod* and *constructMethodCall*. Their definitions are trivial and not shown here. The extraction of the candidate body and parameters for the new method is performed in the same traversal as the legality check. This is a top-down traversal with environment propagation. During descent, the environment is extended with declared variables. When the focus is reached, the legality check is performed. If it succeeds, the free used variables of the focused fragment are determined. These variables are paired with the focused fragment itself, and returned. The legality check itself is defined in the strategy *isLegal*. It fails when the collection of variables that are defined but not declared is non-empty, or when a return statement is recognized in the focus. The replacement of the focus by a new method invocation is defined by the strategy *replaceFocus*. It performs a top-down traversal.

When the focus is found, the new method invocation is generated and the focus is replaced with it.

4. Models of strategies

We have explained what strategy combinators are, and we have shown their utility. Let us now change the point of view, and explain two options for the implementation of the strategy ADTs including the primitives.

4.1 Strategies as functions on a universal term representation

The generic functions needed for strategies have to meet the following requirements. Firstly, they need to be applicable to values of any term type. Secondly, they have to allow for updating in the sense that type-specific behaviour can be enforced. Thirdly, they have to be able to descend into terms. One way to meet these requirements is to rely on a universal representation of terms of algebraic data types. This model can be easily encoded in any functional language although a native language implementation is to be preferred for performance reasons. The notion of a universal representation is relatively simple, and it has also been proposed elsewhere to facilitate the specification of generic functionality, e.g., for the rewriting framework ELAN in [3]. The challenge is to hide the employment of a universal representation to rule out inconsistent representations, and to relieve the programmer of the burden to deal explicitly with representations rather than ordinary values and functions.

The following declarations set up a presentation type *TermRep*, and the ADTs for strategies are defined as functions on *TermRep* wrapped by constructors *MkTP* and *MkTU*:

```
type TypeId          =  String
type ConstrId        =  String
data TermRep         =  TermRep TypeRep ConstrId [TermRep]
data TypeRep         =  TypeRep TypeId [TypeRep]
newtype TP m         =  MkTP (TermRep → m TermRep)
newtype TU m a       =  MkTU (TermRep → m a)
```

Thus, a universal value consists of a type representation (for a potentially parameterized data type), a constructor identifier, and the list of universal values corresponding to the immediate subterms of the encoded term (if any). To mediate between *TermRep* and specific term types, we place members for implosion and explosion in a class *Term*.

```
class Term t where
  explode            ::  t → TermRep
  implode            ::  TermRep → t
```

The instances for a given term type follow a trivial scheme (for details see Appendix II.2). In fact, we extended the DrIFT tool [30] to generate such instances for us (see Section 5). For a faithful universal representation it should hold that explosion can be reversed by implosion. Implosion is potentially a partial operation. One could use the *Maybe* monad for the result to enable recovery from an implosion problem. By contrast, we rule out failure of implosion in the first place by hiding the representation of strategies behind the primitive combinators given below. It is easy to show that all functions on *TermRep* which can be defined in terms of the primitive combinators are implosion-safe.

Let us now look at the definition of the primitive combinators (see Appendix I.2 for the complete code). The combinators *polyTP* and *polyTU* specialize their polymorphic argument to a function on *TermRep*. All the combinators for sequential composition and choice can simply be defined by basically unwrapping the constructors *MkTP* and *MkTU* from each argument and re-wrapping the result.

$$
\begin{array}{lcl}
polyTP\ f & = & MkTP\ f \\
polyTU\ f & = & MkTU\ f \\
\\
unTP\ (MkTP\ f) & = & f \\
unTU\ (MkTU\ f) & = & f
\end{array}
\qquad
\begin{array}{lcl}
seqTP\ f\ g & = & MkTP\ ((unTP\ f)\ `mseq`\ (unTP\ g)) \\
seqTU\ f\ g & = & MkTU\ ((unTP\ f)\ `mseq`\ (unTU\ g)) \\
letTP\ f\ g & = & MkTP\ ((unTU\ f)\ `mlet`\ (\lambda a \rightarrow unTP\ (g\ a))) \\
letTU\ f\ g & = & MkTU\ ((unTU\ f)\ `mlet`\ (\lambda a \rightarrow unTU\ (g\ a))) \\
choiceTP\ f\ g & = & MkTP\ ((unTP\ f)\ `mchoice`\ (unTP\ g)) \\
choiceTU\ f\ g & = & MkTU\ ((unTU\ f)\ `mchoice`\ (unTU\ g))
\end{array}
$$

The combinators for strategy application and updating are defined as follows:

$$
\begin{array}{lcl}
applyTP\ s\ t & = & unTP\ s\ (explode\ t) \gg\!= \lambda t' \rightarrow return\ (implode\ t') \\
applyTU\ s\ t & = & unTU\ s\ (explode\ t) \\
adhocTP\ s\ f & = & MkTP\ (\lambda u \rightarrow \textbf{if}\ applicable\ f\ u \\
 & & \quad \textbf{then}\ f\ (implode\ u) \gg\!= \lambda t \rightarrow return\ (explode\ t) \\
 & & \quad \textbf{else}\ unTP\ s\ u) \\
adhocTU\ s\ f & = & MkTU\ (\lambda u \rightarrow \textbf{if}\ applicable\ f\ u \\
 & & \quad \textbf{then}\ f\ (implode\ u) \\
 & & \quad \textbf{else}\ unTU\ s\ u)
\end{array}
$$

Since strategies are functions on *TermRep*, terms are always first exploded to *TermRep* before the function underlying a strategy can be applied. In the case of a type-preserving strategy, the result of the application also needs to be imploded afterwards. As for update, we use a type test (*cf. applicable*) to check if the given universal value is of the specific type handled by the update. Here, we rely on the fact that we can compare type-representations. If the type test succeeds, the corresponding implosion is performed so that the specific function can be applied. If the type test fails, the generic default strategy is applied.

The primitive traversal combinators are particularly easy to define for this model. Recall that these combinators process in some sense the immediate subterms of a given term. Thus, we can essentially perform list processing. The following code fragment defines a helper to apply a list-processing function on the immediate subterms. We also show the implementation of the primitive *allTP* which employs the standard monadic map function *mapM*.

$$
\begin{array}{lcl}
applyOnKidsTP & :: & Monad\ m \Rightarrow ([\,TermRep] \rightarrow m\ [\,TermRep]) \rightarrow TP\ m \\
applyOnKidsTP\ s & = & MkTP\ (\lambda(TermRep\ sort\ con\ ks) \rightarrow \\
 & & \quad s\ ks \gg\!= \lambda ks' \rightarrow return\ (TermRep\ sort\ con\ ks')) \\
allTP\ s & = & applyOnKidsTP\ (mapM\ (unTP\ s))
\end{array}
$$

4.2 Strategies as rank-2 polymorphic functions

Instead of defining strategies as functions on a universal representation type, we can also define them as a kind of polymorphic functions. Here, we rely on first-order polymorphism [17] combined with class overloading [16]. The following declarations define *TP m* and *TU m a* in terms of universally quantified components of datatype constructors.

$$
\begin{array}{lcl}
\textbf{newtype}\ Monad\ m \Rightarrow TP\ m & = & MkTP\ (\forall t.\ Term\ t \Rightarrow t \rightarrow m\ t) \\
\textbf{newtype}\ Monad\ m \Rightarrow TU\ m\ a & = & MkTU\ (\forall t.\ Term\ t \Rightarrow t \rightarrow m\ a)
\end{array}
$$

As an aside, this form of wrapping is the Haskell approach to deal with rank-2 polymorphism while retaining decidability of type inference [17]. The interesting bit of this model is that the functions which model strategies are not simply universally quantified, but the domain is also constrained to be an instance of the class *Term*. For the present model the term interface looks as follows:

$$
\begin{array}{l}
\textbf{class } \mathit{Update}\ t \Rightarrow \mathit{Term}\ t \textbf{ where} \\
\quad \begin{array}{ll}
\mathit{adhocTP'} :: (\mathit{Monad}\ m, \mathit{Update}\ t') & \Rightarrow\ (t' \to m\ t') \to (t \to m\ t) \to (t' \to m\ t') \\
\mathit{adhocTU'} :: (\mathit{Monad}\ m, \mathit{Update}\ t') & \Rightarrow\ (t' \to m\ a) \to (t \to m\ a) \to (t' \to m\ a) \\
\mathit{allTP'} :: \mathit{Monad}\ m & \Rightarrow\ \mathit{TP}\ m \to t \to m\ t \\
\mathit{oneTP'} :: \mathit{MonadPlus}\ m & \Rightarrow\ \mathit{TP}\ m \to t \to m\ t \\
\mathit{allTU'} :: (\mathit{Monad}\ m, \mathit{Monoid}\ a) & \Rightarrow\ \mathit{TU}\ m\ a \to t \to m\ a \\
\mathit{oneTU'} :: \mathit{MonadPlus}\ m & \Rightarrow\ \mathit{TU}\ m\ a \to t \to m\ a
\end{array}
\end{array}
$$

Thus, some primitives have to be defined for all possible term types. We use primed names because the members are only prototypes which still need to be lifted by wrapping and unwrapping. Specific instances for the traversal primtives are obviously needed when we do not rely on a universal representation type. Specific instances for *adhocTP* and *adhocTU* are needed to model strategy update as a type case [8, 6] via an encoding technique adopted from [29]. For brevity, we omit the class *Update* which is mentioned in the context of *Term*. The members in *Update* are needed to encode *adhocTP* and *adhocTU*. The scheme for all the members in *Term* and *Update* is trivial (and their systematic derivation from the datatypes is routine). We refer to Appendix II.3 and Appendix II.4 for complete sources of the term interface for a simple example. The derivation of the actual primitive combinators is now straightforward. As for sequential composition and choice, the definitions from the previous model carry over. Here are the remaining definitions:

$$
\begin{array}{lcl@{\qquad}lcl}
\mathit{applyTP}\ s\ t & = & (\mathit{unTP}\ s)\ t & \mathit{allTP}\ s & = & \mathit{MkTP}\ (\mathit{allTP'}\ s) \\
\mathit{applyTU}\ s\ t & = & (\mathit{unTU}\ s)\ t & \mathit{oneTP}\ s & = & \mathit{MkTP}\ (\mathit{oneTP'}\ s) \\
\mathit{adhocTP}\ s\ f & = & \mathit{MkTP}\ (\mathit{adhocTP'}\ (\mathit{unTP}\ s)\ f) & \mathit{allTU}\ s & = & \mathit{MkTU}\ (\mathit{allTU'}\ s) \\
\mathit{adhocTU}\ s\ f & = & \mathit{MkTU}\ (\mathit{adhocTU'}\ (\mathit{unTU}\ s)\ f) & \mathit{oneTU}\ s & = & \mathit{MkTU}\ (\mathit{oneTU'}\ s)
\end{array}
$$

As for application, we simply unwrap *MkTP* and *MkTU*. The prototypes from the class *Term* are turned into the proper combinators by wrapping, i.e., the prototypes are explicitely quantified inside *MkTP* and *MkTU*. The present model is not just type-safe as the previous one but the definitions of $\mathit{TP}\ m$ and $\mathit{TU}\ m\ a$ do not need to be hidden any longer.

5. Conclusion

Functional software re-engineering Without appropriate technology large-scale software maintenance projects cannot be done cost-effectively within a reasonable time-span, or not at all [5, 7, 4]. Currently, most (declarative) *re*-technologies are those based on term rewriting frameworks and attribute grammars. There are hardly (published) attempts to employ functional programming for the development of large-scale program transformation systems. One exception is AnnoDomini [9] where SML is used for the implementation of a Y2K tool. The traversal part of AnnoDomini is kept to a reasonable size by a clever normalisation that gets rid of all syntax not relevant to the specific Y2K approach. However, for most re-engineering problems, such a normalisation is not obvious or feasible. Cost-effective solution of such problems requires generic traversal technology that is applicable to the full syntax of the language at hand [4]. In [18], we describe an architecture for functional transformation systems and a corresponding case study concerned with a data expansion problem. This architecture addresses the important issues of scalable parsing and pretty-printing, and employs an approach to generic traversal based on combinators for updatable generalized folds [21]. The functional strategies described in the current paper provide a more lightweight and more generic solution than folds, and can be used instead.

Of course, our techniques are not only applicable to software re-engineering problems, but generally to all areas of language and document processing where traversals are desirable that are both typed and generic. For example, our strategy combinators can be used for XML processing where, in contrast to the approaches presented in [28], document processors can at once be typed and generic.

Generic functional programming Related forms of genericity have been proposed elsewhere. These approaches are rather more complex than ours and they are insufficient for a faithful encoding of the combinators we propose. With intensional and extensional polymorphism [8, 6] one can also encode type-parametric functions where the behaviour is defined via a run-time type case. However, as-is the corresponding systems do not cover algebraic data types, but only products, function space, and basic data types. With polytypic programming (*cf.* PolyP and Generic Haskell [14, 12, 13]), one can define functions by induction on types. However, polytypic functions are not first class citizens: due to the restriction that polytypic parameters are quantified at the top level, polytypic *combinators* cannot be defined. Also, in a polytypic definition, though one can provide fixed ad-hoc cases for specific data types, an *adhoc* combinator which employs a type case, is absent. It may be conceivable that polytypic programming is generalized to cover the functionality of our strategies, but the current paper shows that strategies can be modelled within the type system already available in Haskell.

The origins of functional strategies The term 'strategy' and our conception of generic programming were largely influenced by strategic term rewriting [23, 22, 2, 3, 19]. In particular, the overall idea to define traversal schemes in terms of basic generic combinators like *all* and *one* has been adopted from the untyped language Stratego [25] for strategic term rewriting. Our contribution is that we integrate this idea with typed and higher-order functional programming. In fact, Stratego was not defined with typing in mind. To encode type-unifying traversals, for example, children of a term are processed as inhomogeneous lists. Hence, the resulting traversals are inherently untyped. As an aside, this problem is known from Prolog where one is used to inhomogeneous lists which also arise from generic term destruction and construction via the univ-operator. Furthermore, in Stratego, one is forced to encode type guards for type-specific behaviour by observing constructors. That is, there is no correspondence to our type-dependent update of a strategy in the sense of the *adhoc* combinators. In [19], the first author works out a typeful approach to strategic rewriting. This approach does not rely on a term interface because combinators like *all* and *one* are regarded as true primitives of the designed rewriting calculus. The calculus is very simple but it is not higher-order. In fact, strategies in a higher-order settting have their specific merits. One can, for example, easily deal with effects in a monadic fashion. Our first attempt to rephrase term rewriting strategies for typed higher-order programming was briefly described in [20]. There, we adapted the concept of updatable generalized fold algebras [21] to represent strategies as records. Given a system of datatypes, basic strategy combinators were derived. By contrast, in the present paper the stategy combinators are defined generically where we only rely on a term interface to be instantiated for all term types.

Availability The strategy library *StrategyLib* is available as part of a generic functional programming bundle called *Strafunski* at `http://www.cs.vu.nl/Strafunski`. Complete source of *StrategyLib* are included in Appendix I.3. The bundle also contains an extended version of DrIFT (formerly called Derive [30]) which can be used (pending native support for the *Term* interface) to generate the instances of class *Term* according to the model in Section 4.1 for any given algebraic data type.

References

1. G. Arango, I. Baxter, P. Freeman, and C. Pidgeon. TMM: Software maintenance by transformation. *IEEE Software*, 3(3):27–39, May 1986.
2. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An Overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Proceedings of the International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *ENTCS*, Pont-à-Mousson, France, Sept. 1998. Elsevier Science.
3. P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 2001.
4. M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Generation of Components for Software Renovation Factories from Context-free Grammars. *Science of Computer Programming*, 36(2–3):209–266, 2000.
5. E. Chikofsky and J. C. II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990.
6. K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. *ACM SIGPLAN Notices*, 34(1):301–312, Jan. 1999.
7. A. van Deursen, P. Klint, and C. Verhoef. Research Issues in the Renovation of Legacy Systems. In J. Finance, editor, *Proc. of FASE'99*, volume 1577 of *LNCS*, pages 1–21. Springer-Verlag, 1999.
8. C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *Conference record of POPL'95*, pages 118–129. ACM Press, 1995.
9. P. H. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. B. Sørensen, and M. Tofte. AnnoDomini: From type theory to year 2000 conversion tool. In *Conference Record of POPL'99*, pages 1–14. ACM press, 1999. Invited paper.
10. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
11. *Haskell 98: A Non-strict, Purely Functional Language*, Feb. 1999. `http://www.haskell.org/onlinereport/`.
12. R. Hinze. A generic programming extension for Haskell. In E. Meijer, editor, *Proceedings of the 3rd Haskell Workshop, Paris, France*, Sept. 1999. Technical report, Universiteit Utrecht, UU-CS-1999-28.
13. R. Hinze. A New Approach to Generic Functional Programming. In T. W. Reps, editor, *Conference record of POPL'00*, pages 119–132, Jan. 2000.
14. P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *Conference record of POPL'97*, pages 470–482. ACM Press, 1997.
15. J. Jeuring, editor. *Proc. of WGP'2000, Technical Report, Universiteit Utrecht*, July 2000.

16. M. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 97–136. Springer-Verlag, 1995.
17. M. Jones. First-class polymorphism with type inference. In *Conference record of POPL'97*, pages 483–496, Paris, France, 15–17 Jan. 1997.
18. J. Kort, R. Lämmel, and J. Visser. Functional Transformation Systems. In *9th International Workshop on Functional and Logic Programming*, Benicassim, Spain, July 2000.
19. R. Lämmel. Typed Generic Traversals in S'_γ. Technical report, CWI, Aug. 2001.
20. R. Lämmel and J. Visser. Type-safe Functional Strategies. In *Draft proc. of SFP'00, St Andrews*, July 2000.
21. R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In Jeuring [15], pages 46–59.
22. S. E. M. Clavel, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proc. of RWLW'96*, volume 4 of *ENTCS*, Sept. 1996.
23. L. Paulson. A Higher-Order Implementation of Rewriting. *Science of Computer Programming*, 3(2):119–149, Aug. 1983.
24. E. Visser. Language Independent Traversals for Program Transformation. In Jeuring [15], pages 86–104.
25. E. Visser, Z. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *Proc. of ICFP'98*, pages 13–26, Sept. 1998.
26. P. Wadler. Theorems for Free! In *Proc. of FPCA'89, London*, pages 347–359. ACM Press, New York, Sept. 1989.
27. P. Wadler. The essence of functional programming. In *Conference record of POPL'92*, pages 1–14. ACM Press, 1992.
28. M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? *ACM SIGPLAN Notices*, 34(9):148–159, Sept. 1999. Proceedings of ICFP'99.
29. S. Weirich. Type-safe cast: (functional pearl). *ACM SIGPLAN Notices*, 35(9):58–67, Sept. 2000.
30. N. Winstanley. Derive User Guide, version 1.0. Available at `http://www.dcs.gla.ac.uk/~nww/Derive/`, June 1997.

Appendix I
Source of StrategyLib

This appendix lists the sources of the strategy library *StrategyLib* as available on the *Strafunski* web page `http://www.cs.vu.nl/Strafunski`. The current release (version 1.0) consists of three modules:

- Module *TermRep* for the universal term representation,
- Module *StrategyPrimitives* implementing the strategy primitives according to Section 4.1,
- Module *StrategyLib* offering strategy combinators for the working generic programmer.

Most users will need to import only the top-level module *StrategyLib* into their application modules.

1. Module TermRep

```
{-------------------------------------------------------------------------

                  A LIBRARY OF GENERIC TRAVERSAL COMBINATORS

                  StrategyLib version 1.0, August 10th, 2001

                    Ralf Laemmel                Joost Visser
               CWI & VU, Amsterdam            CWI, Amsterdam

This module is part of a library of generic function combinators, including
combinators for generic traversal. The universal term representation on which
the generic function representation relies, is defined in this module. Most
users will not be concerned with any of the entities defined here.

-------------------------------------------------------------------------}

module TermRep where
```

```
import Monad
import Monoid


--- The universal term representation type -----------------------------------

type TypeId      = String
type ConstrId    = String

data TermRep     = TermRep TypeRep ConstrId [TermRep]
                   deriving (Show,Eq)
data TypeRep     = TypeRep TypeId [TypeRep]
                   deriving (Show,Eq)

typeRep  (TermRep tr _ _)        = tr
constrId (TermRep _ ci _)        = ci
children (TermRep _ _ ks)        = ks


--- The term interface -------------------------------------------------------

class Term t where
  explode       :: t -> TermRep
  implode       :: TermRep -> t
  getTypeRep    :: t -> TypeRep

-- As an aside, one could use a term interface without the member getTypeRep.
-- We do not opt for this omission for simplicity of our scheme.


implodeError t u = error ("Cannot implode to "++t++": "++(show u))

-- We use implodeError in the fall-through cases of the implode members.
-- Strategies are implosion-safe by definition. A different use of the
-- universal representation might however create implosion problems.



--- Instances for basic types and basic type constructors --------------------

    -- String
instance Term String where
  explode str    = TermRep (TypeRep "String" []) str []
  implode (TermRep (TypeRep "String" []) s []) = s
  implode u      = implodeError "String" u
  getTypeRep _   = TypeRep "String" []

    -- List
instance Term a => Term [a] where
  explode xs     = TermRep (TypeRep "List" [elemType]) "" (map explode xs)
                   where elemType = getTypeRep (head xs)
```

```
  implode u@(TermRep listType@(TypeRep "List" [elemType]) _ xs)
                = let xs'            = map implode xs
                      neededElemType = getTypeRep (head xs')
                  in if elemType==neededElemType
                     then xs'
                     else implodeError (show listType) u
  implode u      = implodeError "List" u
  getTypeRep xs  = TypeRep "List" [getTypeRep (head xs)]

    -- Integer
instance Term Integer where
  explode i      = TermRep (TypeRep "Integer" []) (show i) []
  implode (TermRep (TypeRep "Integer" []) i []) = read i
  implode u      = implodeError "Integer" u
  getTypeRep _   = TypeRep "Integer" []

    -- Bool
instance Term Bool where
  explode b      = TermRep (TypeRep "Bool" []) (show b) []
  implode (TermRep (TypeRep "Bool" []) b []) = read b
  implode u      = implodeError "Bool" u
  getTypeRep _   = TypeRep "Bool" []

    -- Maybe
instance Term a => Term (Maybe a) where
  explode mx     = TermRep (TypeRep "Maybe" [elemType]) ""
                       (maybe [] (\x -> [explode x]) mx)
                  where elemType        = getTypeRep (unJust mx)
                        unJust (Just x) = x
  implode u@(TermRep t@(TypeRep "Maybe" [elemType]) _ mx)
                = let xs             = map implode mx
                      neededElemType = (getTypeRep (head xs))
                  in if elemType==neededElemType
                     then case xs of
                            []  -> Nothing
                            [x] -> Just x
                            _   -> implodeError (show t) u
                     else implodeError "Maybe" u
  implode u      = implodeError "Maybe" u
  getTypeRep mx  = TypeRep "Maybe" [getTypeRep (unJust mx)]
                  where unJust (Just x) = x


--------------------------------------------------------------------------------
```

2. Module StrategyPrimitives

```
{-------------------------------------------------------------------------

                  A LIBRARY OF GENERIC TRAVERSAL COMBINATORS

                  StrategyLib version 1.0, August 10th, 2001

                   Ralf Laemmel                Joost Visser
               CWI & VU, Amsterdam            CWI, Amsterdam

This module is part of a library of generic function combinators, including
combinators for generic traversal. The generic function representation that
relies on a universal term representation is defined in this module. The
safety of the generic function combinators is guaranteed by not exporting
the data constructor MkTP.

-------------------------------------------------------------------------}

module StrategyPrimitives (
  TP, TU,                -- Note: MkTP is not exported, MkTU could be exported
  polyTP,
  polyTU,
  unTP,unTU,
  applyTP, applyTU,
  applyOnKidsTU,         -- Note: applyOnKidsTP is not exported
  adhocTP, adhocTU, applicable,
  seqTP, letTP,
  seqTU, letTU,
  choiceTP, choiceTU,
  allTP, oneTP, uniTP, anyTP, someTP,
  allTU, oneTU, uniTU, anyTU, someTU,
  -- And additionally from module TermRep:
  Term
) where

import TermRep
import Monad
import Monoid


--- Strategy representation ------------------------------------------------

newtype TP m     = MkTP (TermRep -> m TermRep)
newtype TU m a   = MkTU (TermRep -> m a)

unTP (MkTP f)    = f
unTU (MkTU f)    = f

polyTP          :: Monad m => (forall t. t -> m t) -> TP m
polyTP f        =  MkTP f
```

```
polyTU          :: Monad m => (forall t. t -> m a) -> TU m a
polyTU f        =  MkTU f


--- Strategy application ---------------------------------------------------

applyTP         :: (Monad m, Term t) => TP m -> t -> m t
applyTP s t     =  do { t' <- unTP s (explode t); return (implode t') }

applyTU         :: (Monad m, Term t) => TU m a -> t -> m a
applyTU s t     =  unTU s (explode t)

applyOnKids     :: Monad m => ([TermRep] -> m [TermRep]) -> TP m
applyOnKids s   =  MkTP (\ (TermRep sort con ks)
                            -> s ks >>= \ks' ->
                               return (TermRep sort con ks'))

applyOnKidsTU   :: ([TermRep] -> m a) -> TU m a
applyOnKidsTU s =  MkTU (\ (TermRep sort con ks) -> s ks)


--- Strategy updating ------------------------------------------------------

adhocTP         :: (Monad m, Term t) => TP m -> (t -> m t) -> TP m
adhocTP s f     =  MkTP (\u -> if applicable f u
                            then do t <- f (implode u)
                                    return (explode t)
                            else (unTP s u) )

adhocTU         :: (Monad m, Term t) => TU m a -> (t -> m a) -> TU m a
adhocTU s f     =  MkTU (\u -> if applicable f u
                            then (f (implode u))
                            else (unTU s u) )

applicable      :: Term a => (a -> b) -> TermRep -> Bool
applicable f u  =  (typeRep u) == (getTypeRep (undefined `withArgType` f))
                   where withArgType :: a -> (a -> x) -> a
                         withArgType a f = a

-- Applicability of a monomorphic function to a universal term representation
-- is defined by comparing the type representation of the argument type of the
-- function with the type representation of the term representation at hand.


--- Deterministic combinators ----------------------------------------------

    -- Type-preserving

seqTP           :: Monad m => TP m -> TP m -> TP m
seqTP f g       =  MkTP ((unTP f) `mseq` (unTP g))
```

```
letTP           :: Monad m => TU m a -> (a -> TP m) -> TP m
letTP f g       =  MkTP ((unTU f) `mlet` (\y -> unTP (g y)))

    -- Type-unifying

seqTU           :: Monad m => TP m -> TU m x -> TU m x
seqTU f g       =  MkTU ((unTP f) `mseq` (unTU g))

letTU           :: Monad m => TU m a -> (a -> TU m x) -> TU m x
letTU f g       =  MkTU ((unTU f) `mlet` (\y -> unTU (g y)))


--- Combinators for partiality and non-determinism ---------------------------

    -- Type-preserving

choiceTP        :: MonadPlus m => TP m -> TP m -> TP m
choiceTP f g    =  MkTP ((unTP f) `mchoice` (unTP g))

    -- Type-unifying

choiceTU        :: MonadPlus m => TU m x -> TU m x -> TU m x
choiceTU f g    =  MkTU ((unTU f) `mchoice` (unTU g))


--- Traversal combinators ----------------------------------------------------

    -- Type-preserving

-- Succeed for all children
allTP           :: Monad m => TP m -> TP m
allTP s         =  applyOnKids (mapM (unTP s))

-- Succeed for one child; don't care about the other children
oneTP           :: MonadPlus m => TP m -> TP m
oneTP s         =  applyOnKids (oneM (unTP s))

-- Succeed for exactly one child; fail for all other children
uniTP           :: MonadPlus m => TP m -> TP m
uniTP s         =  applyOnKids (uniM (unTP s))

-- Succeed for as many children as possible
anyTP           :: MonadPlus m => TP m -> TP m
anyTP s         =  applyOnKids (anyM (unTP s))

-- Succeed for as many children as possible but at least for one
someTP          :: MonadPlus m => TP m -> TP m
someTP s        =  applyOnKids (someM (unTP s))

    -- Helpers
```

```
oneM f []       = mzero
oneM f (x:xs)   = do { x' <- (f x); return (x':xs) }
                  `mplus`
                  do { xs' <- oneM f xs; return (x:xs') }

uniM f []       = mzero
uniM f (x:xs)   = do { x' <- (f x); () <- noneV f xs; return (x':xs) }
                  `mplus`
                  do { () <- noneV f [x]; xs' <- uniM f xs; return (x:xs') }

noneV f []      = return ()
noneV f (x:xs)  = ((f x >>= \_ -> return True) `mplus` (return False))
                  >>= \b -> if b then mzero
                                 else noneV f xs

anyM f []       = return []
anyM f (x:xs)   = do x'  <- ((f x) `mplus` (return x))
                     xs' <-  anyM f xs
                     return (x':xs')

someM f []      = mzero
someM f (x:xs)  = do { x'  <- f x; xs' <- anyM f xs; return (x':xs') }
                  `mplus`
                  do { xs' <- someM f xs; return (x:xs') }

    -- Type-unifying

allTU           :: (Monad m, Monoid a) => TU m a -> TU m a
allTU s         =  applyOnKidsTU (foldM cons mempty)
                   where cons i c = do c' <- (unTU s c)
                                       return (i `mappend` c')

oneTU           :: MonadPlus m => TU m a -> TU m a
oneTU s         =  applyOnKidsTU (foldr cons mzero)
                   where cons c i = (unTU s c) `mplus` i

uniTU           :: MonadPlus m => TU m a -> TU m a
uniTU s         =  applyOnKidsTU (uniTUfold (unTU s))

anyTU           :: (MonadPlus m, Monoid a) => TU m a -> TU m a
anyTU s         =  applyOnKidsTU (foldM (anyTUcons (unTU s)) mempty)

someTU          :: (Monoid a, MonadPlus m) => TU m a -> TU m a
someTU s        =  applyOnKidsTU (someTUfold (unTU s))

    -- Helpers

uniTUfold f []  = mzero
uniTUfold f (x:xs)
  = do { a <- (f x); () <- noneV f xs; return a }
```

```
    `mplus`
    do { () <- noneV f [x]; a <- uniTUfold f xs; return a }

anyTUcons f i c  = do { c' <- f c; return (i `mappend` c') }
                   `mplus`
                   return i

someTUfold f []  = mzero
someTUfold f (x:xs)
                 = do c <- f x
                      i <- (foldM (anyTUcons f) mempty) xs
                      return (i `mappend` c)
                   `mplus`
                   someTUfold f xs


--- Parametric polymorphic prototypes ---------------------------------------

f `mseq` g    = \x -> f x >>= g              -- monadic functional composition
f `mlet` g    = \x -> f x >>= \y -> g y x    -- a kind of monadic let
f `mchoice` g = \x -> (f x) `mplus` (g x)    -- a monadic choice for functions


--- Illustration of type safety ---------------------------------------------

-- The safety of this model of strategies relies on hiding of MkTP. The
-- following term demonstrates unsafe use of the MkTP constructor. We catch a
-- universal representation which arose from a string and we throw it in for
-- the result of a type-preserving function. Clearly, if we attempt to apply
-- the inner type-preserving strategy to a Boolean constant, then we end up
-- with an implosion error.

brokenTerm       = applyTU
                     (MkTU (\u -> applyTP (MkTP (const (Just u))) True))
                     "BREAK"

-- If the polyTP constructor function is used instead, the resulting term
-- is not typable:
--
-- untypableTerm = applyTU
--                   (MkTU (\u -> applyTP (polyTP (const (return u))) True))
--                   "BREAK"
--
-- Thus, by not exporting MkTP, safety is guaranteed.

------------------------------------------------------------------------------
```

3. Module StrategyLib

```
{-------------------------------------------------------------------------------

                  A LIBRARY OF GENERIC TRAVERSAL COMBINATORS

                  StrategyLib version 1.0, August 10th, 2001

                   Ralf Laemmel                Joost Visser
               CWI & VU, Amsterdam            CWI, Amsterdam

This module is part of a library of generic function combinators, including
combinators for generic traversal. Most users should import only this module
into their application modules.

-------------------------------------------------------------------------------}

module StrategyLib (
  module StrategyPrimitives,
  module StrategyLib
) where

import StrategyPrimitives
import Monad
import Monoid


--- Traversal control ------------------------------------------------------

    -- Type-preserving combinators

-- Identity strategy
identity        :: Monad m => TP m
identity        =  polyTP return

-- Failure strategy
failTP          :: MonadPlus m => TP m
failTP          =  polyTP (const mzero)

-- Lift a function to a strategy type with failure as default
monoTP          :: (Term a, MonadPlus m) => (a -> m a) -> TP m
monoTP f        =  adhocTP failTP f

-- Always succeed by a catch-all
tryTP           :: MonadPlus m => TP m -> TP m
tryTP s         =  s `choiceTP` identity

-- Exhaustive repeated application
repeatTP        :: MonadPlus m => TP m -> TP m
repeatTP s      =  tryTP (seqTP s (repeatTP s))
```

```
-- Place a guard on a strategy
ifthenTP        :: Monad m => TP m -> TP m -> TP m
ifthenTP f g    =  (f `seqTU` void) `letTP` (const g)

-- Perform a test without changing terms
testTP          :: Monad m => TP m -> TP m
testTP f        =  f `ifthenTP` identity

-- Negation by failure
notTP           :: MonadPlus m => TP m -> TP m
notTP s         =  ((s `ifthenTU` (build True)) `choiceTU` (build False))
                   `letTP` \b ->
                   if b then failTP else identity

    -- Type-unifying combinators

build           :: Monad m => a -> TU m a
build a         =  polyTU (const (return a))

void            :: Monad m => TU m ()
void            =  build ()

failTU          :: MonadPlus m => TU m x
failTU          =  polyTU (const mzero)

monoTU          :: (Term a, MonadPlus m) => (a -> m b) -> TU m b
monoTU f        =  adhocTU failTU f

tryTU           :: (MonadPlus m, Monoid a) => TU m a -> TU m a
tryTU s         =  s `choiceTU` (build mempty)

ifthenTU        :: Monad m => TP m -> TU m b -> TU m b
ifthenTU f g    =  (f `seqTU` void) `letTU` (const g)

testTU          :: Monad m => TU m a -> TP m
testTU f        =  f `letTP` (const identity)

notTU           :: MonadPlus m => TU m a -> TP m
notTU s         =  choiceTU (s `letTU` const (build True)) (build False)
                   `letTP` \b ->
                   if b then failTP else identity

comb            :: Monad m => (a -> b -> c) -> TU m a -> TU m b -> TU m c
comb o s s'     =  s  `letTU` \a ->
                   s' `letTU` \b ->
                   build (o a b)

appendTU        :: (Monoid a, MonadPlus m) => TU m a -> TU m a -> TU m a
appendTU f g    =  comb mappend (tryTU f) (tryTU g)

afterTU         :: Monad m => (a -> b) -> TU m a -> TU m b
```

```
afterTU f s     =  s `letTU` \a -> build (f a)


--- Traversal schemes ----------------------------------------------------

    -- Type-preserving combinators

{-
   For performance and uniformity reasons, anyTP and someTP are
   primitives, but they could have been defined as follows:

anyTP           :: MonadPlus m => TP m -> TP m
anyTP s         =  allTP (tryTP s)

someTP          :: MonadPlus m => TP m -> TP m
someTP s        =  (testTP (notTP (allTP (notTP s)))) `seqTP` (anyTP s)

-}

td              :: Monad m => TP m -> TP m
td s            =  s `seqTP` (allTP (td s))

bu              :: Monad m => TP m -> TP m
bu s            =  (allTP (bu s)) `seqTP` s

stoptd          :: MonadPlus m => TP m -> TP m
stoptd s        =  s `choiceTP` (allTP (stoptd s))

oncetd          :: MonadPlus m => TP m -> TP m
oncetd s        =  s `choiceTP` (oneTP (oncetd s))

oncebu          :: MonadPlus m => TP m -> TP m
oncebu s        =  (oneTP (oncebu s)) `choiceTP` s

    -- Type-unifying combinators

crush           :: (MonadPlus m, Monoid a) => TU m a -> TU m a
crush s         =  comb mappend (tryTU s) (allTU (crush s))

collect         :: MonadPlus m => TU m [a] -> TU m [a]
collect s       =  crush s

select          :: MonadPlus m => TU m a -> TU m a
select s        =  s `choiceTU` (oneTU (select s))

selectenv       :: MonadPlus m => e -> (e -> TU m e) -> (e -> TU m a) -> TU m a
selectenv e s' s = s' e `letTU` \e' ->
                   (s e) `choiceTU` (oneTU (selectenv e' s' s))
```

```
--- Fixpoint traversal -------------------------------------------------------

reduce s         = repeatTP(someTP (reduce s) `choiceTP` s)

outermost s      = repeatTP(oncetd s)

innermost' s     = repeatTP(oncebu s)

innermost s      = allTP(innermost s)
                   `seqTP`
                   (tryTP (s `seqTP` (innermost s)))

------------------------------------------------------------------------------
```

Appendix II
Example instantiations of the term interface

In this appendix, we illustrate the term interface underlying the two models of functional strategies as discussed in Section 4. As we pointed out earlier, a strategic programmer is not obliged to deliver this term interface him- or herself. In the Strafunski bundle, an updated version of the DrIFT tool is available which can be used to generate the instances for the *Term* class from the data types (for the model in Section 4.1).

Below, we use a minimal system of data types rather than a Java grammar to illustrate the schemes. The system is minimal in the sense that it is the smallest sytem that involves (i) mutual recursion, (ii) constructors with zero, one, and multiple children, and (iii) a basic data type (*Integer* in this case).

1. A sample system of data types

```
--- A simple system of mutually recursive datatypes --------------------------

module Datatypes where

data SortA = SortA1 SortB | SortA2 deriving Show
data SortB = SortB Integer SortA   deriving Show

-------------------------------------------------------------------------------
```

2. Term interface for universal representation

```
--- Term interface for TwoSorts example based on universal representation ----

module TermInterface where

import TermRep
import Datatypes


--- Term interface for SortA -------------------------------------------------

instance Term SortA where

  explode a@(SortA1 b) = TermRep (getTypeRep a) "SortA1" [explode b]
  explode a@SortA2     = TermRep (getTypeRep a) "SortA2" []

  implode (TermRep t "SortA1" [b])
    | t == getTypeRep (undefined::SortA) = SortA1 (implode b)
  implode (TermRep t "SortA2" [])
    | t == getTypeRep (undefined::SortA) = SortA2

  getTypeRep _ = TypeRep "SortA" []

-- We use the applications of getTypeRep in the definition of implode to
-- reuse the definition of getTypeRep rather than to repeat the type
-- representation in the guards of the equations. We omit cases for implosion
-- errors. Such errors cannot occur with strategies. They can occur if a
-- programmer uses the universal representation type for other purposes than
-- strategic programming.

--- Term interface for SortA -------------------------------------------------

instance Term SortB where

  explode b@(SortB i a) = TermRep (getTypeRep b) "SortB" [explode i,explode a]

  implode (TermRep t "SortB" [i,a])
    | t == getTypeRep (undefined::SortB) = SortB (implode i) (implode a)

  getTypeRep _ = TypeRep "SortB" []

------------------------------------------------------------------------------
```

3. Rank-2 term interface: class *Update*

```
{- The type case for type-safe update ----------------------------------------

   Updating works as follows. For brevity, let us consider adhocTP' in the
   Term class only (it works quite the same for TU). The overloaded function
   adhocTP' takes a polymorphic function p and a monomorphic function
   m :: s -> m s for a specific term type s.  In the Term instance for s
   the function adhocTP' is defined in terms of a member sTP from the Update
   class where sTP is specific to s. In fact, the trick is that the specific
   member sTP records the type s. The result type of adhocTP' p m is again
   polymorphic. When adhocTP' p m is ultimately applied to a term of some type
   t, then the instance for t determines the interpretation of sTP p m. If t
   and the original s are the same, then m wins, otherwise we resort to p.
   In summary, the instances for the Update class encode n*n cases in a type
   case to decide if an update for s has to be applied to a given term where
   n is the number of data types in the system at hand.

------------------------------------------------------------------------------}

module Update where

import Datatypes


-- Class declaration with closed world assumption ----------------------------

class Update t where
  integerTP :: Monad m => (t -> m t) -> (Integer -> m Integer) -> (t -> m t)
  boolTP    :: Monad m => (t -> m t) -> (Bool    -> m Bool)    -> (t -> m t)
  sortaTP   :: Monad m => (t -> m t) -> (SortA   -> m SortA)   -> (t -> m t)
  sortbTP   :: Monad m => (t -> m t) -> (SortB   -> m SortB)   -> (t -> m t)
  integerTU :: Monad m => (t -> m a) -> (Integer -> m a)       -> (t -> m a)
  boolTU    :: Monad m => (t -> m a) -> (Bool    -> m a)       -> (t -> m a)
  sortaTU   :: Monad m => (t -> m a) -> (SortA   -> m a)       -> (t -> m a)
  sortbTU   :: Monad m => (t -> m a) -> (SortB   -> m a)       -> (t -> m a)


-- Instances for type case ---------------------------------------------------

instance Update Integer where
  integerTP _ f = f
  boolTP f _    = f
  sortaTP f _   = f
  sortbTP f _   = f
  integerTU _ f = f
  boolTU f _    = f
  sortaTU f _   = f
  sortbTU f _   = f
```

```
instance Update Bool where
  integerTP f _ = f
  boolTP _ f    = f
  sortaTP f _   = f
  sortbTP f _   = f
  integerTU f _ = f
  boolTU _ f    = f
  sortaTU f _   = f
  sortbTU f _   = f


instance Update SortA where
  integerTP f _ = f
  boolTP f _    = f
  sortaTP _ f   = f
  sortbTP f _   = f
  integerTU f _ = f
  boolTU f _    = f
  sortaTU _ f   = f
  sortbTU f _   = f


instance Update SortB where
  integerTP f _ = f
  boolTP f _    = f
  sortaTP f _   = f
  sortbTP _ f   = f
  integerTU f _ = f
  boolTU f _    = f
  sortaTU f _   = f
  sortbTU _ f   = f

--------------------------------------------------------------------------------
```

4. Rank-2 term interface: class *Term*

```
--- Term interface for TwoSorts example based on rank-2 types ---------------

module TermInterface where

import Datatypes
import Update
import Monad
import Monoid
import StrategyPrimitives


--- The instances for Integer, Bool, SortA, SortB ----------------------------

instance Term Integer where
  adhocTP' = integerTP
  adhocTU' = integerTU
  allTP' _ = return
  oneTP' _ = const mzero
  allTU' _ = const (return mempty)
  oneTU' _ = const mzero


instance Term Bool where
  adhocTP' = boolTP
  adhocTU' = boolTU
  allTP' _ = return
  oneTP' _ = const mzero
  allTU' _ = const (return mempty)
  oneTU' _ = const mzero


instance Term SortA where

  adhocTP' = sortaTP

  adhocTU' = sortaTU

  allTP' f (SortA1 b) = applyTP f b >>= \b' -> return (SortA1 b')
  allTP' f SortA2     = return SortA2

  oneTP' f (SortA1 b) = applyTP f b >>= \b' -> return (SortA1 b')
  oneTP' f SortA2     = mzero

  allTU' f (SortA1 b) = applyTU f b
  allTU' f SortA2     = return mempty

  oneTU' f (SortA1 b) = applyTU f b
  oneTU' f SortA2     = mzero
```

```
instance Term SortB where

  adhocTP' = sortbTP

  adhocTU' = sortbTU

  allTP' f (SortB i a) = applyTP f i >>= \i' ->
                         applyTP f a >>= \a' ->
                         return (SortB i' a')

  oneTP' f (SortB i a) = (applyTP f i >>= \i' -> return (SortB i' a)) `mplus`
                         (applyTP f a >>= \a' -> return (SortB i a'))

  allTU' f (SortB i a) = applyTU f i >>= \i' ->
                         applyTU f a >>= \a' ->
                         return (i' `mappend` a')

  oneTU' f (SortB i a) = (applyTU f i) `mplus`
                         (applyTU f a)

--------------------------------------------------------------------------------
```

Table of Contents

1 Introduction . . . 1
2 A strategy library . . . 2
2.1 Strategy types and application . . . 3
2.2 Strategy construction . . . 3
2.3 Sequential composition . . . 4
2.4 Partiality and non-determinism . . . 4
2.5 Traversal combinators . . . 5
2.6 Some defined combinators . . . 5
3 Application: Refactoring . . . 6
3.1 The *extract method* refactoring . . . 6
3.2 Design of the algorithm . . . 7
3.3 Implementation with strategies . . . 7
4 Models of strategies . . . 10
4.1 Strategies as functions on a universal term representation . . . 10
4.2 Strategies as rank-2 polymorphic functions . . . 11
5 Conclusion . . . 12

Bibliography **13**

References **14**

I Source of StrategyLib **16**
1 Module TermRep . . . 16
2 Module StrategyPrimitives . . . 19
3 Module StrategyLib . . . 24

II Example instantiations of the term interface **28**
1 A sample system of data types . . . 28
2 Term interface for universal representation . . . 29
3 Rank-2 term interface: class *Update* . . . 30
4 Rank-2 term interface: class *Term* . . . 32