

Typed Combinators for Generic Traversal

Ralf Lämmel^{1,2} and Joost Visser¹

¹ CWI, Kruislaan 413, NL-1098 SJ Amsterdam

² Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam

Email: (Ralf.Laemmel|Joost.Visser)@cwi.nl

WWW: [http://www.cwi.nl/~\(ralf|jvisser\)/](http://www.cwi.nl/~(ralf|jvisser)/)

Phone: +31 20 592 (4090|4266) and Fax: +31 20 592 4199

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.

Keywords: Genericity, traversal, combinators, program transformation

1 Introduction

Our domain of interest is program transformation in the context of software re-engineering [4, 2, 3]. 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 [3, 15].

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.¹ We show how these strategies can be modelled inside the functional language Haskell², and we present a strategy combinator library that includes traversal combinators.

¹ We use the term *generic* in the general sense of type- or syntax-independent, not in the stricter senses of parametric polymorphism or polytypism. In fact, the genericity of functional strategies goes beyond these stricter senses.

² Throughout the paper we use Haskell 98 [10], unless stated otherwise.

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);
```

For this fragment, the traversal should return the list ["_name", "amount"] of variables on use sites.

Using the techniques to be presented in this paper, the desired traversal can be modelled with a function of the following type:

$$\text{collectUseVars} \quad :: \quad TU \text{ Maybe } [String]$$

Here, $TU \text{ Maybe } [String]$ is the type of *type-unifying* generic functions which map terms of any type to a list of *Strings*. The *Maybe* monad is used to model partiality. In general, a function f of type $TU \ m \ a$ can be applied to a term of *any* type to yield a result of type a (of a monadic type $m \ a$ to be precise). Besides type-unifying strategies, we will later encounter so-called *type-preserving* strategies where input and output type coincide.

The definition of *collectUseVars* can be based on a simple and completely generic traversal scheme of the following name and type:

$$\text{collect} \quad :: \quad MonadPlus \ m \Rightarrow \ TU \ m \ [a] \rightarrow \ TU \ m \ [a]$$

The strategy combinator *collect* maps a type-unifying strategy intended for identification of collectable entities in a node to a type-unifying strategy performing the actual collection over the entire syntax tree. This traversal combinator is included in our library. We can use the combinator in the following manner to collect Java variables on use sites:

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

The non-generic, monomorphic function *useVar* identifies variable names in Java expressions. To make it suitable as an argument to *collect*, it is turned into a type-unifying generic function by feeding it to the combinator *monoTU*. The resulting traversal *collectUseVars* can be applied to any kind of Java program fragment, and it will return the variables identified by *useVar*. Note that the constructor functions *Just* and *Nothing* are used to construct a value of the *Maybe* datatype to represent the list of identified variables.

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 general, functional strategies can be employed in a scalable way to construct programs that operate on large syntaxes. 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 paper 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 universal term representation, and an implementation that relies on rank-2 polymorphism and type case. The paper is concluded in Section 5.

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. In the running text we will provide definitions of a number of defined strategies, including some traversal schemes.

2.1 Strategy types and application

There are two kinds of strategies. Firstly, the ADT $TP\ m$ 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 $TU\ m\ a$ 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 [22] to deal with effects in strategies such as state passing or non-determinism. Also note that we do not apply strategies

Strategy types (opaque)	
data <i>Monad m</i>	$\Rightarrow TP\ m = \dots\ abstract$
data <i>Monad m</i>	$\Rightarrow TU\ m\ a = \dots\ abstract$
Strategy application	
<i>applyTP</i> :: (<i>Monad m</i> , <i>Term t</i>)	$\Rightarrow TP\ m \rightarrow t \rightarrow m\ t$
<i>applyTU</i> :: (<i>Monad m</i> , <i>Term t</i>)	$\Rightarrow TU\ m\ a \rightarrow t \rightarrow m\ a$
Strategy construction	
<i>polyTP</i> :: <i>Monad m</i>	$\Rightarrow (\forall x. x \rightarrow m\ x) \rightarrow TP\ m$
<i>polyTU</i> :: <i>Monad m</i>	$\Rightarrow (\forall x. x \rightarrow m\ a) \rightarrow TU\ m\ a$
<i>ad hocTP</i> :: (<i>Monad m</i> , <i>Term t</i>)	$\Rightarrow TP\ m \rightarrow (t \rightarrow m\ t) \rightarrow TP\ m$
<i>ad hocTU</i> :: (<i>Monad m</i> , <i>Term t</i>)	$\Rightarrow TU\ m\ a \rightarrow (t \rightarrow m\ a) \rightarrow TU\ m\ a$
Sequential composition	
<i>seqTP</i> :: <i>Monad m</i>	$\Rightarrow TP\ m \rightarrow TP\ m \rightarrow TP\ m$
<i>letTP</i> :: <i>Monad m</i>	$\Rightarrow TU\ m\ a \rightarrow (a \rightarrow TP\ m) \rightarrow TP\ m$
<i>seqTU</i> :: <i>Monad m</i>	$\Rightarrow TP\ m \rightarrow TU\ m\ a \rightarrow TU\ m\ a$
<i>letTU</i> :: <i>Monad m</i>	$\Rightarrow TU\ m\ a \rightarrow (a \rightarrow TU\ m\ b) \rightarrow TU\ m\ b$
Choice	
<i>choiceTP</i> :: <i>MonadPlus m</i>	$\Rightarrow TP\ m \rightarrow TP\ m \rightarrow TP\ m$
<i>choiceTU</i> :: <i>MonadPlus m</i>	$\Rightarrow TU\ m\ a \rightarrow TU\ m\ a \rightarrow TU\ m\ a$
Traversal combinators	
<i>allTP</i> :: <i>Monad m</i>	$\Rightarrow TP\ m \rightarrow TP\ m$
<i>oneTP</i> :: <i>MonadPlus m</i>	$\Rightarrow TP\ m \rightarrow TP\ m$
<i>allTU</i> :: (<i>Monad m</i> , <i>Monoid a</i>)	$\Rightarrow TU\ m\ a \rightarrow TU\ m\ a$
<i>oneTU</i> :: <i>MonadPlus m</i>	$\Rightarrow TU\ m\ a \rightarrow TU\ m\ a$

Fig. 1. Primitive strategy combinators.

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.

The strategy application combinators serve to turn a generic functional strategy into a non-generic function which can be applied to a term of a specific type. Recall that the introductory example is a type-unifying traversal with the result type [*String*]. It can be applied to a given Java class declaration *myClassDecl* of type *ClassDeclaration* as follows:

```
applyTU collectUseVars myClassDecl :: Maybe [String]
```

Prerequisite for this code to work is that an instance of the class *Term* is available for *ClassDeclaration*. This issue will be taken up in Section 4.

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. *ad hocTP* and *ad hocTU*). 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:

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

The type-preserving strategy $\mathit{identity}$ denotes the generic (and monadic) identity function. The type-unifying strategy $\mathit{build}\ a$ denotes the generic function which returns a regardless of the input term. As a consequence of parametricity [21], there are no further ways to inhabit the argument types of polyTP and polyTU , unless we rely on a specific instance of m (see failTU below).

The second way of strategy construction, i.e., with the *ad hoc* 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:

$$\begin{array}{ll} \mathit{gnot} & :: \text{Monad } m \Rightarrow TP\ m \\ \mathit{gnot} & = \mathit{ad hocTP}\ \mathit{identity}\ (\mathit{return}\ \circ\ \mathit{not}) \end{array}$$

The strategy gnot is applicable to terms of any type. It will behave like $\mathit{identity}$ 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:

$$\begin{array}{ll} g \circ f & = \lambda x \rightarrow g\ (f\ x) \\ f\ \mathit{'mseq'}\ g & = \lambda x \rightarrow f\ x \ggg g \\ f\ \mathit{'mlet'}\ g & = \lambda x \rightarrow f\ x \ggg \lambda y \rightarrow g\ y\ x \end{array}$$

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.

Recall that the *poly* combinators could be used to lift an ordinary parametric polymorphic function to a strategy. We can not just use *poly* to lift the prototypes for sequential composition because they are function *combinators*. For this reason, we supply the combinators for sequential composition as primitives of the ADTs, and we postpone their definition to Section 4.

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:

$$\begin{aligned} comb & \quad :: \text{Monad } m \Rightarrow (a \rightarrow b \rightarrow c) \rightarrow TU\ m\ a \rightarrow TU\ m\ b \rightarrow TU\ m\ c \\ comb\ o\ s\ s' & = s\ 'letTU'\ \lambda a \rightarrow s'\ 'letTU'\ \lambda b \rightarrow build\ (o\ a\ b) \end{aligned}$$

Thus, the result of the first strategy argument *s* is bound to the variable *a*. Then, the result of the second strategy argument *s'* is bound to *b*. Finally, *a* and *b* are combined with the operator *o*, and the result is returned by the *build* combinator which was defined Section 2.2.

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.

$$f\ 'mchoice'\ g \quad = \lambda x \rightarrow (f\ x)\ 'mplus'\ (g\ x)$$

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

$$\begin{aligned} failTU & \quad :: \text{MonadPlus } m \Rightarrow TU\ m\ x \\ failTU & = polyTU\ (const\ mzero) \\ monoTU & \quad :: (\text{Term } a, \text{MonadPlus } m) \Rightarrow (t \rightarrow m\ a) \rightarrow TU\ m\ a \\ monoTU\ f & = adhocTU\ failTU\ f \\ tryTU & \quad :: (\text{MonadPlus } m, \text{Monoid } a) \Rightarrow TU\ m\ a \rightarrow TU\ m\ a \\ tryTU\ s & = s\ 'choiceTU'\ (build\ mempty) \end{aligned}$$

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.

Recall that the *monoTU* combinator was used in the introductory example to turn the non-generic, monomorphic function *useVar* into a type-unifying strategy. This strategy will fail when applied to any type other than *Expression*.

2.5 Traversal combinators

A challenging facet of strategies is that they might descend into 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 [20] 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.

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

Note that the *comb* combinator is used to combine the result of *s* on the current node with the result of crushing the subterms. The *tryTU* combinator is used to recover from possible failure of *s*. In the introductory example, this comes down to recovery from failure of *monoTU useVar* at non-*Expression* nodes, and at nodes of type *Expression* for which *useVar* returns *Nothing*.

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. Here are some representatives of the category for the control of strategies:

$$\begin{aligned}
\textit{repeatTP} &:: \textit{MonadPlus } m \Rightarrow \textit{TP } m \rightarrow \textit{TP } m \\
\textit{repeatTP } s &= \textit{tryTP } (\textit{seqTP } s (\textit{repeatTP } s)) \\
\textit{ifthenTP} &:: \textit{Monad } m \Rightarrow \textit{TP } m \rightarrow \textit{TP } m \rightarrow \textit{TP } m \\
\textit{ifthenTP } f \ g &= (f \ \textit{'seqTU'} (\textit{build } ())) \ \textit{'letTP'} (\textit{const } g) \\
\textit{notTP} &:: \textit{MonadPlus } m \Rightarrow \textit{TP } m \rightarrow \textit{TP } m \\
\textit{notTP } s &= ((s \ \textit{'ifthenTU'} (\textit{build } \textit{True})) \ \textit{'choiceTU'} (\textit{build } \textit{False})) \\
&\quad \textit{'letTP'} \lambda b \rightarrow \textit{if } b \ \textit{then } \textit{failTP} \ \textit{else } \textit{identity} \\
\textit{afterTU} &:: \textit{Monad } m \Rightarrow (a \rightarrow b) \rightarrow \textit{TU } m \ a \rightarrow \textit{TU } m \ b \\
\textit{afterTU } f \ s &= s \ \textit{'letTU'} \lambda a \rightarrow \textit{build } (f \ a)
\end{aligned}$$

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*):

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

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 is to 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. As will be demonstrated in the upcoming section, traversal schemes like these can serve as building blocks for program transformations.

3 Application: Refactoring

Refactoring [9] is the process of step-wise improving the internal structure of a software system without altering its external behaviour. The *extract method refactoring* [9, 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

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.

$$\begin{aligned}
\text{typed_free_vars} &:: (\text{MonadPlus } m, \text{Eq } v) \\
&\Rightarrow [(v, t)] \rightarrow \text{TU } m [v] \rightarrow \text{TU } m [(v, t)] \rightarrow \text{TU } m [(v, t)] \\
\text{typed_free_vars } env \text{ getvars declvars} \\
&= \text{afterTU } (\text{flip } \text{appendMap } env) (\text{tryTU } \text{declvars}) \text{'letTU' } \lambda env' \rightarrow \\
&\quad \text{choiceTU } (\text{afterTU } (\text{flip } \text{selectMap } env') \text{ getvars}) \\
&\quad (\text{comb } \text{diffMap } (\text{allTU } (\text{typed_free_vars } env' \text{ getvars } \text{declvars})) \\
&\quad \quad (\text{tryTU } \text{declvars}))
\end{aligned}$$

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

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.

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 [19]. 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. It is not constructed by reusing one of the defined traversal combinators from our library, but directly in terms of the primitive combinator *allTU*. 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 (cf. *allTU*). 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 analyses 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

```

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 vds
        declVarsMeth (MethodDecl _ _ (FormalParams fps) _) = return fps
freeUseVars env = afterTU nubMap (typed_free_vars env (monoTU useVar) declVars)
freeDefVars env = afterTU nubMap (typed_free_vars env (monoTU defVar) declVars)

```

Fig. 3. Instantiations of the generic free variable algorithm for Java.

them to instantiate the generic free variable collector to construct *freeUseVars*, and *freeDefVars*.

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 some options for the implementation of the strategy ADTs including the primitives. Recall that functional 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. The first model we discuss uses a universal term representation. The second model employs rank-2 polymorphism with type case.

```

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)

```

Fig. 4. Implementation of the *extract method* refactoring.

4.1 Strategies as functions on a universal term representation

One way to meet the requirements on functional strategies is to rely on a universal representation of terms of algebraic datatypes. Such a representation can easily be constructed in any functional language in a straightforward manner. The challenge is to hide the employment of the 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 representation type *TermRep*, and the ADTs for strategies are defined as functions on *TermRep* wrapped by datatype 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). The strategy ADTs are made opaque by simply not exporting the constructors *MkTP* and *MkTU*. 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

```

```

explode (Identifier i) = TermRep (TypeRep "Expr" [] "Identifier" [explode i])
implode (TermRep _ "Identifier" [i]) = Identifier (implode i)

```

The instances for a given term type follow a trivial scheme, as illustrated by the two sample equations for Java *Identifiers*. In fact, we extended the DrIFT tool [25] 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 defined below. It would be easy to prove that all functions on *TermRep* which can be defined in terms of the primitive combinators are implosion-safe.

The combinators *polyTP* and *polyTU* specialize their polymorphic argument to a function on *TermRep*. Essentially, the combinators for sequential composition and choice are also defined by specialisation of the corresponding prototypes *mseq*, *mlet*, and *mchoice*. In addition, we need to unwrap the constructors *MkTP* and *MkTU* from each argument and to re-wrap the result.

```

polyTP f = MkTP f      seqTP f g = MkTP ((unTP f) 'mseq' (unTP g))
polyTU f = MkTU f    seqTU f g = MkTU ((unTP f) 'mseq' (unTU g))
unTP (MkTP f) = f    letTP f g = MkTP ((unTU f) 'mlet' (λ a → unTP (g a)))
unTU (MkTU f) = f   letTU f g = MkTU ((unTU f) 'mlet' (λ a → unTU (g a)))
                       choiceTP f g = MkTP ((unTP f) 'mchoice' (unTP g))
                       choiceTU f g = MkTU ((unTU f) 'mchoice' (unTU g))

```

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

```

applyTP s t = unTP s (explode t) ≫ λt' → return (implode t')
applyTU s t = unTU s (explode t)
ad hocTP s f = MkTP (λu → if applicable f u
                      then f (implode u) ≫ λt → return (explode t)
                      else unTP s u)
ad hocTU s f = MkTU (λu → if applicable f u
                       then f (implode u)
                       else unTU s u)

```

As for application, terms are always first exploded to *TermRep* before the function underlying a strategy can be applied. This is because strategies are functions on *TermRep*. 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. For brevity, we omit the definition of *applicable* but it simply compares 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 directly employs the standard monadic map function *mapM*.

```

applyOnKidsTP :: Monad m => ([TermRep] → m [TermRep]) → TP m
applyOnKidsTP s = MkTP (λ(TermRep sort con ks) →
                       s ks ≫ λks' → return (TermRep sort con ks'))
allTP s          = applyOnKidsTP (mapM (unTP s))

```

4.2 Strategies as rank-2 polymorphic functions with type case

Instead of defining strategies as functions on a universal representation type, we can also define them as a kind of polymorphic functions being directly applicable to terms of the algebraic datatypes. But, since strategies can be passed as arguments to strategy combinators, we need to make use of *rank-2 polymorphism*³. The following declarations define *TP m* and *TU m a* in terms of universally quantified components of datatype constructors. This form of wrapping is the Haskell approach to deal with rank-2 polymorphism while retaining decidability of type inference [14].

```

newtype Monad m => TP m    = MkTP (∀t. Term t => t → m t)
newtype Monad m => TU m a = MkTU (∀t. Term t => t → m a)

```

³ Rank-2 polymorphism is not part of Haskell 98, but available in the required form as an extension of the Hugs implementation.

Note 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*. The following model-specific term interface provides traversal and ad-hoc primitives to meet the other requirements on strategies.

```

class Update t => Term t where
  allTP' :: Monad m           => TP m -> t -> m t
  oneTP'  :: MonadPlus m      => TP m -> t -> m t
  allTU'  :: (Monad m, Monoid a) => TU m a -> t -> m a
  oneTU'  :: MonadPlus m      => TU m a -> t -> m a
  adhocTP' :: (Monad m, Update t') => (t' -> m t') -> (t -> m t) -> (t' -> m t')
  adhocTU' :: (Monad m, Update t') => (t' -> m a) -> (t -> m a) -> (t' -> m a)

```

We use primed names because the members are only rank-1 prototypes which still need to be lifted by wrapping and unwrapping. The term interface is instantiated by defining the primitives for all possible term types.

The definitions of the traversal primitives are as simple as the definitions of the *implode* and *explode* functions for the previous model. They are not shown for brevity. To define *adhocTP'* and *adhocTU'* for each datatype, an additional technique is needed: we model strategy update as a type case [7, 5]. The instances of the *Update* class, mentioned in the context of class *Term*, implement this type case via an encoding technique for Haskell inspired by [24]. In essence, this technique involves two members *dUpdTP* and *dUpdTU* in the *Update* class for each datatype *d*. These members for *d* select their second argument in the instance for *d*, and default to their first argument in all other instances.

Given the rank-1 prototypes, the derivation of the actual rank-2 primitive combinators is straightforward:

```

applyTP s t = (unTP s) t           allTP s = MkTP (allTP' s)
applyTU s t = (unTU s) t           oneTP s = MkTP (oneTP' s)
adhocTP s f = MkTP (adhocTP' (unTP s) f)  allTU s = MkTU (allTU' s)
adhocTU s f = MkTU (adhocTU' (unTU s) f)  oneTU s = MkTU (oneTU' s)

```

Note that application does not involve conversion with *implode* and *explode*, as in the previous model, but only unwrapping of the rank-2 polymorphic function. As for sequential composition, choice, and the *poly* combinators, the definitions from the previous model carry over.

4.3 Trade-offs and alternatives

The model relying on a universal term representation is simple and does not rely on more than parametric polymorphism and class overloading. It satisfies extensibility in the sense that for each new datatype, one can provide a new instance of *Term* without invalidating previous instances. The second model is slightly more involved. But it is more appealing in that no conversion is needed, because strategies are simply functions on the datatypes themselves, instead of on a representation of them. However, extensibility is compromised, as the employed coding scheme for type cases involves a closed world assumption. That is,

the encoding technique for type case requires a class *Update* which has members for each datatype. Note that these trade-offs are Haskell-specific. In a different language, e.g., a language with built-in type case, strategies would be supported via different models. In fact, a simple language extension could support strategies directly.

Regardless of the model, it is intuitively clear that a full traversal visiting all nodes should use time linear in the size of the term, assuming a constant node-processing complexity. Both models expose this behaviour. However, if a traversal stops somewhere, no overhead for non-traversed nodes should occur. The described universal representation is problematic in this respect since the non-traversed part below the stop node will have to be imploded before the node can be processed. Thus, we suffer a penalty linear in the number of non-traversed nodes. Similarly, implosion is needed when a strategy is applied which involves an ad-hoc update. This is because a universal representation has to be imploded before a non-generic function can be applied on a node of a specific datatype. Short of switching to the second model, one can remedy these performance problems by adopting a more involved universal representation. The overall idea is to use dynamic typing [1] and to do stepwise explosion by need, that is, only if the application of a traversal primitive requires it.

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 [4, 6, 3]. Currently, declarative *re*-technologies are usually 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 [8] where SML is used for the implementation of a Y2K tool. The traversal part of AnnoDomini is kept to a reasonable size by a specific normalisation that gets rid of all syntax not relevant for this Y2K approach. In general, re-engineering requires generic traversal technology that is applicable to the full syntax of the language at hand [3]. In [15], we describe an architecture for functional transformation systems and a corresponding case study concerned with a data expansion problem. The 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 [17]. The functional strategies described in the present 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 type-safe generic traversal is desirable. For example, our strategy combinators can be used for XML processing where, in contrast to the approaches presented in [23], document processors can at once be typed and generic.

Generic functional programming Related forms of genericity have been proposed elsewhere. These approaches are not just more complex than ours, but they are even insufficient for a faithful encoding of the combinators we propose. With intensional and extensional polymorphism [7, 5] 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 [12, 11]), 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 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 a language like Haskell without type-system extensions.

The origins of functional strategies The term ‘strategy’ and our conception of generic programming were largely influenced by strategic term rewriting [18, 20, 16]. 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 [20] 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.

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 the report version of this paper which can also be downloaded from the Strafunski web page. The bundle also contains an extended version of DrIFT (formerly called Derive [25]) which can be used (pending native support for strategies) to generate the instances of class *Term* according to the model in Section 4.1 for any given algebraic data type.

References

1. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, Apr. 1991.
2. G. Arango, I. Baxter, P. Freeman, and C. Pidgeon. TMM: Software maintenance by transformation. *IEEE Software*, 3(3):27–39, May 1986.
3. M. Brand, M. 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.
4. E. Chikofsky and J. C. II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990.
5. K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. *ACM SIGPLAN Notices*, 34(1):301–312, Jan. 1999.

6. A. 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.
7. C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *Conference record of POPL'95*, pages 118–129. ACM Press, 1995.
8. 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.
9. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
10. *Haskell 98: A Non-strict, Purely Functional Language*, Feb. 1999. <http://www.haskell.org/onlinereport/>.
11. 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.
12. P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *Conference record of POPL'97*, pages 470–482. ACM Press, 1997.
13. J. Jeuring, editor. *Proc. of WGP'2000, Technical Report, Universiteit Utrecht*, July 2000.
14. M. Jones. First-class polymorphism with type inference. In *Conference record of POPL'97*, pages 483–496, Paris, France, 15–17 Jan. 1997.
15. J. Kort, R. Lämmel, and J. Visser. Functional Transformation Systems. In *9th International Workshop on Functional and Logic Programming*, Benicassim, Spain, July 2000.
16. R. Lämmel. Typed Generic Traversals in S'_{γ} . Technical Report SEN-R0122, CWI, Aug. 2001.
17. R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In Jeuring [13], pages 46–59.
18. L. Paulson. A Higher-Order Implementation of Rewriting. *Science of Computer Programming*, 3(2):119–149, Aug. 1983.
19. E. Visser. Language Independent Traversals for Program Transformation. In Jeuring [13], pages 86–104.
20. E. Visser, Z. Benaïssa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *Proc. of ICFP'98*, pages 13–26, Sept. 1998.
21. P. Wadler. Theorems for Free! In *Proc. of FPCA'89, London*, pages 347–359. ACM Press, New York, Sept. 1989.
22. P. Wadler. The essence of functional programming. In *Conference record of POPL'92*, pages 1–14. ACM Press, 1992.
23. 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.
24. S. Weirich. Type-safe cast: (functional pearl). *ACM SIGPLAN Notices*, 35(9):58–67, Sept. 2000.
25. N. Winstanley. Derive User Guide, version 1.0. Available at <http://www.dcs.gla.ac.uk/~nww/Derive/>, June 1997.