Strategic Programming

João Saraiva and José Nuno Macedo

Department of Informatics & HASLab/INESC TEC
Universidade do Minho, Braga, Portugal
`saraiva@di.uminho.pt`

Recursive functions are known to be suited to express computations on recursive data types, such as lists and trees. For example, we can define in Haskell the algebraic data type *Tree* modeling a binary tree where integer values are stored in its leaf nodes,

```
data Tree = Leaf Int
          | Fork Tree Tree
```

and straightforwardly implement a recursive function that computes its minimum value:

```
treemin :: Tree → Int
treemin (Leaf i)   = i
treemin (Fork l r) = min (treemin l) (treemin r)
```

The use of algebraic data types and pattern matching makes the program concise, and easy to understand and manipulate.

When we consider bigger data types, however, we lose this conciseness and elegance! Let us consider the (sub)language of **let** expressions as incorporated in most functional languages, including Haskell. Next, we show two examples of programs in the **let** language, which corresponds to correct *Haskell* code.

```
program = let a = b + 3              nestedprogram = let a = b + 3
              c = 2                                      c = 2
              b = c × 3 − c                              w = let c = a × b
          in  (a + 7) × c                                    in  c × b + 0
                                                         b = c × 3 − c
                                                     in  (a + 7) × c + w
```

We can easily define the abstract (syntactic) structure of **let** expressions by the following heterogeneous data type *Let*.

```
data Let = Let List Exp

data List = Assign    String Exp List
          | NestedLet String Let List
          | EmptyList

data Exp = Add   Exp Exp
         | Mul   Exp Exp
         | Neg   Exp
         | Const Int
         | Var   String
```

Let us consider that we wish to *optimize* this Haskell fragment by eliminating addition with 0. Obviously, can define a recursive function that performs the desired transformation.

```
optimize :: Let → Let
optimize (Let l e) = Let (optimizeList l) (optimizeExp e)
optimizeList :: List → List
optimizeList (Assign s e l)      = Assign s (optimizeExp e) (optimizeList l)
optimizeList (NestedLet s l ls) = NestedLet s (optimize l) (optimizeList l)
optimizeList EmptyList           = EmptyList
optimizeExp :: Exp → Exp
optimizeExp (Add e (Const 0)) = e
optimizeExp (Add (Const 0) e) = e
optimizeExp (Add e1 e2)        = Add (optimizeExp e1) (optimizeExp e2)
optimizeExp (Mul e1 e2)        = Mul (optimizeExp e1) (optimizeExp e2)
optimizeExp (Neg e)            = Neg (optimizeExp e)
optimizeExp (Const i)          = Const i
optimizeExp (Var s)            = Var s
```

Looking in detail to this definition we see that only in two alternative definitions of function *optimizeExp* the function does useful work: it applies the arithmetic expression rule. All other alternatives contain *boring* code, just expressing the recursion on *Let* trees. In fact, our optimization is defined by three functions - *optimize*, *optimizeList*, and *optimizeExp* - expressed on eleven alternative definitions. This number of definitions correspond to the three data types that contain nine constructors: The additional two alternatives are the ones where interesting work is being done.

In this example, we have considered a tiny subset of the Haskell language: **let** expressions. Things get much more complex when we consider large scale language transformations. Let us consider that we wish to perform the above optimization in full Haskell code. The algebraic data type defining the abstract syntax of Haskell[1] contains 116 constructors across 30 data types. As a consequence, performing such a simple transformation would (probably) require the definition of 30 functions containing more than one hundred *boring* alternatives!

Obviously, we should **not write** such complex and large recursive functions. Thus, we need a different programming abstraction to express large scale language transformations. In this context, strategic term rewriting is an extremely suitable formalism [1, 2], since it provides a solution that just defines the work to be done in the constructors (tree nodes) of interest, and "ignores" all the others. The key idea underlying strategic programming is the separation of problem-specific ingredients of traversal functionality (i.e., type specific basic actions) and reusable traversal schemes (i.e., traversal control).

[1] available in the Haskell library `https://hackage.haskell.org/package/haskell-src-1.0.4/docs/Language-Haskell-Syntax.html`

Let us return to our **let** optimization problem. In the style of strategic programming we define we a constructor/node specific rewriting function expressing the desired transformation/optimization:

```
expr :: Exp → Exp
expr (Add e (Const 0)) = e
expr (Add (Const 0) e) = e
expr e                 = e
```

where the first two alternatives define the optimization: when either of the sub-expressions of an *Add* expression is the constant 0, then it returns the other sub-expression. The last alternative defines the *default* behavior for all other cases, returning the original expression.

Having defined the type specific transformation, we use *strategic functions* to apply it while traversing the tree according to a recursion scheme (*i.e.*, a strategy). Next, we show how we express it in strategic programming:[2] the *apply* function, applies the strategy *full_td* that performs a *full-top-down* traversal on the input tree t. Thus, the worker function *expr* is applied to tree nodes in a full top down traversal scheme.

```
optimize t = apply (full_td expr) t
```

In this particular case we used a full top down traversal scheme. Strategic programming, however, offer many of typical traversal schemes such as full bottom up, once bottom up, innermost, etc.

As we can see in the strategic solution of *optimize*, in strategic programming our solutions focus on the nodes/constructors where work needs to be performed, while omitting the nodes/constructors where no work needs to be performed. As result, strategic programming provides concise and elegant solutions to express large scale language engineering tasks.

To define generic strategies we need a generic mechanism to traverse and transform trees. Next, we introduce the zipper data structure that offers a generic mechanism to navigate on tree data structures and perform generic tree transformations. After that we show how to express strategic programming via Zippers. Then, we show the use of the developed Ztrategic library in practice, by giving examples of the use of most of its strategies.

1 Functional Zippers

Zippers were originally conceived by Huet [3] to represent a tree together with a subtree that is the focus of attention. During a computation, the focus may move left, up, down or right within the tree. Generic manipulation of a zipper is provided through a set of predefined functions that allow access to all of the nodes of the tree for inspection or modification.

[2] We omit here some details of this definition. They will be defined and explained in the remainder of this text.

In order to illustrate the use of zippers, let us again consider again the binary leaf-tree *Tree* data type. Next, we define a tree t_1 of this type:

$$
\begin{aligned}
& t_1 :: \mathit{Tree} \\
& t_1 = \mathit{Fork}\ (\mathit{Leaf}\ 3) \\
& \qquad\qquad (\mathit{Fork}\ (\mathit{Leaf}\ 2) \\
& \qquad\qquad\qquad\quad (\mathit{Leaf}\ 4))
\end{aligned}
$$

When we consider t_1 as a whole, we notice that each of the subtrees occupies a certain location in tree. In fact, a location of a subtree may be represented by the subtree itself and by the rest of the tree. Thus, the rest of the tree can be viewed as the context of that subtree.

One of the possible ways to represent this context is as a path from the top of the tree to the node which is the focus of attention (where the subtree has its root). This means that it is possible to use contexts and subtrees to define any position in the tree, while creating a setting where navigation is provided by the contextual information of a path applied to a tree.

For example, if we wish to put the focus in *Leaf* 2, its context in tree t_1 is:

$$
\begin{aligned}
& \mathit{tree} = \mathit{Fork}\ (\mathit{Leaf}\ 3) \\
& \qquad\qquad\quad (\mathit{Fork}\ \otimes \\
& \qquad\qquad\qquad\quad (\mathit{Leaf}\ 4))
\end{aligned}
$$

where $\otimes$ points the exact location where *Leaf* 4 occurs. One possible way to represent this context consists in defining a path from the top of the tree to the position which is the focus of attention. To reach *Leaf* 2 in tree t_1, we need to go right (down the right branch) and then left (down the left one). In practice, this means that given the tree t_1, then the path [right, left] would suffice in indicating the location in the tree we want to focus our attention on.

This is precisely the main concept behind zippers. Using the idea of having pairs of information composed by paths and trees, we may represent any positions in a tree and, better yet, this setting allows an easy navigation as we only have to remove parts of the path if we want to go to the top of the tree, or add information if we want to go further down.

Using this idea, we may represent contexts as instances of the following datatype:

$$
\begin{aligned}
\mathbf{data}\ \mathit{Cxt}\ a = {} & \mathit{Top} \\
\mid {} & L\ (\mathit{Cxt}\ a)\ a \\
\mid {} & R\ a\qquad (\mathit{Cxt}\ a)
\end{aligned}
$$

A value $L\ c\ t$ represents the left part of a branch of which the right part is t and whose parent has context c. Similarly, a value $R\ t\ c$ represents the right part of a branch of which the left part is t and whose parent has context c. The value *Top* represents the top of the tree. Returning to our example, we may represent the context of *Leaf* 2 in the tree as:

$$
\begin{aligned}
& \mathit{context} :: \mathit{Cxt}\ \mathit{Tree} \\
& \mathit{context} = L\ (R\ (\mathit{Leaf}\ 3)\ \mathit{Top})\ (\mathit{Leaf}\ 4)
\end{aligned}
$$

which is the same as saying that the focus results from going to the left in a subtree whose right side is *Leaf* 4. This subtree is obtained from going to the right from the top tree, whose left side is *Leaf* 3.

Having defined the context of a subtree, we may define a location on a tree as one of its subtrees together with its context:

```
type Loc a = (a, Cxt a)
```

We are now ready to present the definition of useful functions that manipulate locations on binary trees. First, we can define a function that goes down the left branch of a tree:

```
left :: Loc Tree → Loc Tree
left (Fork l r, c) = (l, L c r)
```

This function takes a tuple with a tree and a context (a *Loc*) and creates a new tuple, where the left side of the tree becomes the new tree, and a new context is created, extending the previous one with a data constructor L whose right sides becomes the right side of the previous tree. Similarly, we introduce a function that goes down the right branch:

```
right :: Loc Tree → Loc Tree
right (Fork l r, c) = (r, R l c)
```

Going up in the tree corresponds to removing "directions" from the path. The function *parent* goes up on a tree location:

```
parent :: Loc Tree → Loc Tree
parent (t, L c r) = (Fork t r, c)
parent (t, R l c) = (Fork l t, c)
```

This function has two alternatives: when it gets as argument a location whose context is a left side (L), then it creates a new location where the right side of the context is moved to a new tree and the left side is kept as the context (c). The second alternative does the opposite when the context of the location is a right side (R).

Finally, we define a function *zipper* that creates a tree location from a tree (where we define an empty context):

```
zipper :: Tree → Loc Tree
zipper t = (t, Top)
```

and we also define another function that extracts a tree from a tree location, by simply taking the first element of the location pair:

```
value :: Loc Tree → Tree
value = fst
```

Having defined a zipper-based function to navigate in binary-trees, we can now focus the attention on subtree *Leaf* 2 of t_1 as follows:

```
leafWith2 :: Loc Tree
leafWith2 = left (right (zipper t1))
```

Once the subtree that is the focus of our attention is reached, we may perform several actions on it. One such action would be to edit that subtree: we may want, for example, to decrement its leaf value by one. Using the zipper functions defined so far, this is expressed as follows:

```
edit = let (Leaf v, cxt) = leafWith2
           subtree'      = (Leaf (v - 1), cxt)
       in  value (parent (parent subtree'))
```

with *edit* producing the result:

```
edited = Fork (Leaf 3)
              (Fork (Leaf 1)
                    (Leaf 4))
```

In conclusion, we have just presented an instance of a zipper for the *Tree* data type, and we showed how to navigate and to express transformations on such trees. In the next section, we present a generic zipper *Haskell* library that offers this functionality to any data type.

2 Generic Zippers

Generic zippers are available as a *Haskell* library [4], which works for both homogeneous and heterogeneous data types. In order to show the expressiveness of this library we shall start by navigating in the homogeneous *Tree* data type. After that we will show how generic zippers can navigate in heterogeneous ones.

In order to navigate on tree t_1, first, the data type needs to be an instance of the *Data* and *Typeable*. Any data type that has an instance of the *Data* and *Typeable* type classes can be navigated using generic zippers. This is, for example, the case of Haskell pre-defined list data type. In the case of our *Tree* data type we need to define those instances. Next, we use the deriving primitive to produce them.

```
data Tree = Leaf Int
          | Fork Tree Tree
          deriving (Data, Typeable)
```

Now, we use the generic zipper library function

toZipper :: $Data\ a \Rightarrow a \rightarrow$ Zipper a.

to transform a tree into a zipper:

```
t1' = Zipper Tree
t1' = toZipper t1
```

This function produces an aggregate data structure which is easy to traverse and update. For example, we may move the focus of attention on t_1' from the topmost node to the leaf containing the number 3, as follows:

```
leafWith3 :: Tree
leafWith3 = (fromJust . getHole . fromJust . down') t1'
```

where the zipper library function

```
down' :: Zipper a → Maybe (Zipper a)
```

goes down to the leftmost (immediate) child of a node, whereas function

```
getHole :: Typeable b ⇒ Zipper a → Maybe b
```

extracts the node under focus from a zipper. The library also includes a function down to go to the rightmost (immediate) child of a node, and functions up, left, and right to navigate in the directions their names indicate.

All these functions wrap the result inside a *Maybe* data type to make them total. To simplify our example we are unwrapping it using the function *fromJust*.[3] The function *leafWith3* navigates from the tree's root via the shortest path to the desired leaf. Next, we navigate to that leaf after visiting all nodes in the tree as expressed by *fullVisit* in a compositional style. To avoid the use of *fromJust* and also to obtain a more natural *left-to-right* writing/reading of the navigation on trees, we can adopt an applicative functional style as expressed by *fullVisit'*, where the predefined *monadic binding* function (>>=) allows the value returned from the first computation to influence the choice of the next one.

```
fullVisit :: Tree
fullVisit = (fromJust . getHole .
             fromJust . left     .
             fromJust . up       .
             fromJust . left     .
             fromJust . down     .
             fromJust . down     .
             fromJust . down'    ) t1'
```

```
fullVisit' :: Tree
fullVisit' = fromJust  $  down' t1'
                          >>= down
                          >>= down
                          >>= left
                          >>= up
                          >>= left
                          >>= getHole
```

Obviously, not all tree paths are valid. Next, we present an unfeasible one:

```
noPath :: Maybe Tree
noPath = (getHole . fromJust . left . fromJust . down') t1'
```

Generic zippers provide an uniform way to navigate in heterogeneous data structures, without the need of distinguishing which type of nodes it is traversing.

[3] Actually, we are not really checking for totality, otherwise we would have to test each function call against a set of possible results.

Let us consider that the *Let* data type has been updated to derive instances of classes *Data* and *Typeable*.

Show example of navigation on let

2.1 Transformations with Zippers

Generic Zippers do not only support navigation on heterogeneous trees (as showed before), but they also support generic transformations. In fact, the zipper library contains functions for the transformation of the data structure being traversed. The function

$trans :: GenericT \to \mathsf{Zipper}\ a \to \mathsf{Zipper}\ a$

applies a generic transformation to the node the zipper is currently pointing to; while

$transM :: GenericM\ m \to \mathsf{Zipper}\ a \to m\ (\mathsf{Zipper}\ a)$

applies a generic monadic transformation.

In order to show a zipper-based transformation, let us consider that we wish to increment a constant in a **let** expression. We begin by defining a function *incConstant* that increments constants, and use the generic function *mkT* (from the generics library [5]) to generalize this type-specific function to all types:

```
incConstant :: Exp → Exp
incConstant (Const n) = Const (n + 1)
incConstantG :: GenericT
incConstantG = mkT incConstant
```

This function has type *GenericT* (meaning *Generic Transformation*) that is the required type of *trans*. To transform the assignment $c = 2$ (in p) to $c = 3$ we just have to navigate to the desired constant and then apply *trans*, as follows[4]:

```
incrC :: Maybe (Zipper Let)
incrC = do t2 ← down'  t1
           t_3 ← down' t2
           t_4 ← right  t_3
           t_5 ← right  t_4
           t_6 ← down' t_5
           t_7 ← right  t_6
           return (trans incConstantG t_7)
```

3 Zipper-based Strategic Programming

The two key ingredients for expressing strategic term re-writing are: a generic data structure traversal mechanism and a generic data structure transformation

[4] Given that *Maybe* is a monad, this time we avoided the repeated use of *fromJust* by writing the code in a monadic style.

mechanism. As we showed in Section 2, generic zippers provide both mechanisms and, thus, they are the key ingredient to express strategic programming as we will show in this Section.

Before we discuss how to model strategies via zippers, let us return to our **let** optimization problem. Next, we present the type specific transformation function where we return a *Maybe* result to express transformations that may fail (that is, do nothing).

```
expr :: Exp → Maybe Exp
expr (Add e (Const 0)) = Just e
expr (Add (Const 0) e) = Just e
expr e                 = Just e
```

This function applies to *Exp* nodes only. To express our **let** optimization, however, we need a generic mechanism that traverses *Let* trees, applying this function when visiting *Add* expressions. This is where strategic term rewriting comes to the rescue: It provides recursion patterns (*i.e.*, strategies) to traverse the (generic) tree, like, for example, top-down or bottom-up traversals. It also includes functions to apply a node specific rewriting function (like *expr*) according to a given strategy. Next, we show the strategic solution of our optimization where *expr* is applied to the input tree in a full top-down strategy. This is a Type Preserving (TP) transformation since the input and result trees have the same type:

```
optimize :: Zipper Let → Maybe (Zipper Let)
optimize t = applyTP (full_tdTP step) t
  where step = idTP `adhocTP` expr
```

We have just presented our first (zipper-based) strategic function. Here, *step* is a transformation to be applied by function applyTP to all nodes of the input tree *t* (of type Zipper *Let*) using a full top-down traversal scheme (function full_tdTP). The rewrite step behaves like the identity function (idTP) by default with our *expr* function to perform the type-specific transformation, and the adhocTP combinator joining them into a single function. In this solution, we clearly see that the traversal function full_tdTP needs to navigate heterogeneous trees, as it is the case of *Let* expressions. Moreover, the traversal functions also need to apply transformations at specific nodes, as it is the case of function *expr*. Thus, we will rely again on zippers to provide such a generic transformation and tree-walk mechanism to embed strategic programming in *Haskell*. In fact, our strategic combinators work with zippers as in the definition of *optimize*.

In the next section we describe how data structure transformations are directly supported by zippers. Then, in Section 3.1 we present in detail the embedding of strategies using the generic zipper mechanism.

3.1 Strategic Programming with Zippers

In this section we introduce Ztrategic, our embedding of strategic programming using generic zippers. The embedding directly follows the work of Laemmel and Visser on the Strafunski library [6, 7]. Before we present the powerful and reusable strategic functions providing control on tree traversals, such as top-down, bottom-up, innermost, etc., let us show some simple basic combinators that work at the zipper level, and are the building blocks of our embedding.

We start by defining a function that elevates a transformation to the zipper level. In other words, we define how a function that is supposed to operate directly on one data type is converted into a transformation that operates on a zipper.

```
zTryApplyM   :: (Typeable a, Typeable b) ⇒ (a → Maybe b) → TP c
zTryApplyM f = transM (join . cast . f . fromJust . cast)
```

The definition of zTryApplyM relies on transformations on zippers, thus reusing the generic zipper library *transM* function. To build a valid transformation for the *transM* function, we use the *cast* :: $a \to$ *Maybe* b function, that tries to cast a given data from type a to type b. In this case, we use it to cast the data the zipper is focused on into a type our original transformation f can be applied to. Then, function f is applied, and its result is cast back to its original type. Should any of these casts, or the function f itself, fail, the failure propagates and the resulting zipper transformation will also fail. The use of the monadic version of the zipper generic transformation guarantees the handling of such partiality. It should be noticed that failure can occur in two situations: the type cast fails when the types do not match. Moreover, the function f fails when the function itself dictates that no change is to be applied. Signaling failure in the application of transformations is important for strategies where a transformation is applied once, only.

zTryApplyM returns a TP c, in which TP is a type for specifying Type-Preserving transformations on zippers, and c is the type of the zipper. For example, if we are applying transformations on a zipper built upon the *Let* data type, then those transformations are of type TP *Let*.

```
type TP a = Zipper a → Maybe (Zipper a)
```

Very much like Strafunski, we introduce the type TU m d for Type-Unifying operations, which aim to gather data of type d into the data structure m.

```
type TU m d = (forall a . Zipper a → m d)
```

For example, to collect in a list all the defined names in a **let** expression, the corresponding type-unifying strategy would be of type TU [ ] *String*. We will present such a transformation and implement it later in this section.

Next, we define a combinator to compose two transformations, building a more complex zipper transformation that tries to apply each of the initial transformations in sequence, skipping transformations that fail.

```
adhocTP :: Typeable a ⇒ TP e → (a → Maybe a) → TP e
adhocTP f g z = maybeKeep f (zTryApplyM g) z
```

The adhocTP function receives transformations f and g as parameters, as well as zipper z. It converts g, which is a simple (*i.e.* non-zipper) *Haskell* function, into a zipper transformation, and uses the auxiliary function *maybeKeep* to try to apply the two transformations to the zipper z, ignoring if it fails.

Next, we use adhocTP, written as an infix operator, which combines the zipper function failTP with our basic transformation *expr* function:

```
step = failTP `adhocTP` expr
```

Thus, we do not need to express type-specific transformations as functions that work on zippers. It is the use of zTryApplyM in adhocTP that transforms a *Haskell* function (*expr* in this case) into a zipper one, hidden from these definitions.

The function failTP is a pre-defined transformation that always fails and idTP is the identity transformation that always succeeds. They provide the basis for construction of complex transformations through composition. We omit here their simple definitions. The functions we have presented already allow the definition of arbitrarily complex transformations for zippers. Such transformations, however, are always applied on the node the zipper is focusing on. Let us consider a combinator that navigates in the zipper.

```
allTPright :: TP a → TP a
allTPright f z = case right z of
                   Nothing → return z
                   Just r   → fmap (fromJust . left) (f r)
```

This function is a combinator that, given a type-preserving transformation f for zipper z, will attempt to apply f to the node that is located to the right of the node the zipper is pointing to. To do this, the zipper function right is used to try to navigate to the right; if it fails, we return the original zipper. If it succeeds, we apply transformation f and then we navigate left again. There is a similar combinator allTPdown but navigating downwards and then upwards.

With all these tools at our disposal, we can define generic traversal schemes by combining them. Next, we define the traversal scheme used in the function *opt*, which we defined at the start of the section. This traversal scheme navigates through the whole data structure, in a top-down approach.

```
full_tdTP :: TP a → TP a
full_tdTP f = allTPdown (full_tdTP f) `seqTP`
              allTPright  (full_tdTP f) `seqTP` f
```

We skip the explanation of the seqTP operator as it is relatively similar to the adhocTP operator we have described before, albeit simpler; we interpret this as a sequence operator. This function receives as input a type-preserving transformation f, and (reading the code from right to left) it applies it to the focused node itself, then to the nodes below the currently focused node, then to the nodes to the right of the focused node. To apply this transformation to the nodes below the current node, for example, we use the allTPdown combinator we mentioned above, and we recursively apply full_tdTP f to the node below. The same logic applies in regards to navigating to the right.

We can define several traversal schemes similar to this one by changing the combinators used, or their sequence. For example, by inverting the order in which the combinators are sequenced, we define a bottom-up traversal. By using different combinators, we can define choice, allowing for partial traversals in the data structure. We previously defined a rewrite strategy where we use full_tdTP to define a full, top-down traversal, which is not ideal. Because we intend to optimize *Exp* nodes, changing one node might make it possible to optimize the node above, which would have already been processed in a top-down traversal. Instead, we define a different traversal scheme, for repeated application of a transformation until a fixed point is reached:

```
innermost :: TP a → TP a
innermost s = repeatTP (once_buTP s)
```

We omit the definitions of once_buTP and repeatTP as they are similar to the presented definitions. The combinator repeatTP applies a given transformation repeatedly, until a fixed point is reached, that is, until the data structure stops being changed by the transformation. The transformation being applied repeatedly is defined with the once_buTP combinator, which applies s once, anywhere on the data structure. When the application once_buTP fails, repeatTP understands a fixed point is reached. Because the once_buTP bottom-up combinator is used, the traversal scheme is innermost, since it prioritizes the innermost nodes. The pre-defined outermost strategy uses the once_tdTP combinator instead. The full API of our zipper-based strategic library is presented in Fig. 1.

4 Ztrategic Library: Examples

Let us return to our **let** running example. Next, we show a **let** fragment that we will use to show the power of strategic programming.

```
example = let a = b − 16
              c = 8
              w = let   y = let q = 3 + 4
                            in q
                        z = 0 + a + 0
                  in z + b
              b = (c + 3) − c
          in c + a − w
```

Looking at this **let** program we can see that several arithmetic rules can be applied, namely, we can remove the two additions with zero in the definition of z, and we can perform the addition of 3 with 4 in the nested definition on q.

To make our example more interesting, we consider more arithmetic rules that we may use to optimize let expressions. In Figure 2, we present the rules given in [8].

Strategy Types
```
type TP a = Zipper a → Maybe (Zipper a)
type TU m d = (forall a . Zipper a → m d)
```
Primitive Strategies
```
idTP     :: TP a
constTU  :: d → TU m d
failTP   :: TP a
failTU   :: TU m d
tryTP    :: TP a → TP a
repeatTP :: TP a → TP a
```
Strategic Construction
```
monoTP   :: (a → Maybe b) → TP e
monoTU   :: (a → m d) → TU m d
monoTPZ  :: (a → Zipper e → Maybe b) → TP e
monoTUZ  :: (a → Zipper e → m d) → TU m d
adhocTP  :: TP e → (a → Maybe b) → TP e
adhocTU  :: TU m d → (a → m d) → TU m d
adhocTPZ :: TP e → (a → Zipper e → Maybe b)
            → TP e
adhocTUZ :: TU m d → (a → Zipper c → m d)
            → TU m d
```
Composition / Choice
```
seqTP    :: TP a → TP a → TP a
choiceTP :: TP a → TP a → TP a
seqTU    :: TU m d → TU m d → TU m d
choiceTU :: TU m d → TU m d → TU m d
```
Traversal Combinators
```
allTPright :: TP a → TP a
oneTPright :: TP a → TP a
allTUright :: TU m d → TU m d
allTPdown  :: TP a → TP a
oneTPdown  :: TP a → TP a
allTUdown  :: TU m d → TU m d
```
Traversal Strategies
```
full_tdTP :: TP a → TP a
full_buTP :: TP a → TP a
once_tdTP :: TP a → TP a
once_buTP :: TP a → TP a
stop_tdTP :: TP a → TP a
stop_buTP :: TP a → TP a
innermost :: TP a → TP a
outermost :: TP a → TP a
full_tdTU :: TU m d → TU m d
full_buTU :: TU m d → TU m d
once_tdTU :: TU m d → TU m d
once_buTU :: TU m d → TU m d
stop_tdTU :: TU m d → TU m d
stop_buTU :: TU m d → TU m d
```

Fig. 1: Full Ztrategic API

$$add(e, const(0)) \rightarrow e \tag{1}$$
$$add(const(0), e) \rightarrow e \tag{2}$$
$$add(const(a), const(b)) \rightarrow const(a+b) \tag{3}$$
$$sub(e1, e2) \rightarrow add(e1, neg(e2)) \tag{4}$$
$$neg(neg(e)) \rightarrow e \tag{5}$$
$$neg(const(a)) \rightarrow const(-a) \tag{6}$$

Fig. 2: Optimization Rules

In our definition of the function *expr*, we already defined rewriting rules for optimizations 1 and 2. We can trivially extend this definition to consider rules 3 through 6:

```
expr :: Exp → Maybe Exp
expr (Add e (Const 0))          = Just e                -- (1)
expr (Add (Const 0) t)          = Just t                -- (2)
expr (Add (Const a) (Const b))  = Just (Const (a + b))  -- (3)
expr (Sub a b)                = Just (Add a (Neg b))   -- (4)
expr (Neg (Neg f))            = Just f                 -- (5)
expr (Neg (Const n))          = Just (Const (−n))      -- (6)
expr _                        = Just e                 -- default
```

Having expressed all rewriting rules in function *expr*, now we need to use our strategic solution *optimize* that navigates in the tree while applying the rules. Next, we consider several strategies to apply this type specific transformation, which produce different optimizations.

Full Top Down Strategy: This strategy traverses the full tree while applying the transformation to the nodes where the type specific transformation holds.

Next, we show the function *optimizeFTP* that applies the optimizations defined by *expr* using the full_tdTP strategy.

```
optimizeFTP :: Zipper Let → Maybe (Zipper Let)
optimizeFTP t = applyTP (full_tdTP step) t
  where step = idTP ‘adhocTP‘ expr
```

The strategic function *optmizeFTP* combines all the steps of our library. We define an auxiliary function *step*, which is the composition of the idTP default succeeding strategy with *expr*, the optimization function; we compose them with adhocTP. Our resulting Type-Preserving strategy will be full_tdTP *step*, which applies *step* to the zipper repeatedly while fully traversing the tree in a top down scheme. We apply this strategy using the function applyTP :: TP c → Zipper c → *Maybe* (Zipper c), which effectively applies a strategy to a zipper.

To make it easier to apply this function to our examples, we make this function work on *Let* trees, and not on the Zipper of these trees.

```
optimizationFTD :: Let → Let
optimizationFTD t = fromZipper (fromJust (applyTP (full_tdTP step) (toZipper t)))
  where step = idTP ‘adhocTP‘ expr
```

When we apply *optimize* that uses the strategy full_tdTP to the *letExample* input, we get the following result:

```
letResult = let a = b + 16
                c = 8
                w = let y = let q = 7
                            in q
                        z = 0 + a
                    in z + b
                b = (c + 3) − c
            in c + a − w
```

The full top down traversal scheme does not perform all possible optimizations. In fact, it was only able to eliminate a single occurrence of the two additions with 0. This happens because when we apply the rule that eliminates addition with 0 to the subtree $0 + a + 0$, the first alternative definition in *expr* matches the input. As a result, the first pattern instantiates $0 + a$ with the (pattern) variable e and 0 with *Const* 0. This line of code returns e as result, thus returning the $0 + a$ that pattern matched with e (without performing further optimization on e).

Innermost Strategy: A innermost strategy applies the type specific transformation to the tree until a fix point is reached. That is to say, until no transformation can be applied and it returns as result the given input tree.

In our **let** example, in order to guarantee that all possible optimizations are applied to the input, we need to use an innermost traversal scheme. Thus, our optimization is correctly expressed as:

```
optmizeIM  :: Zipper Let → Maybe (Zipper Let)
optmizeIM t = applyTP (innermost step) t
  where step = failTP ‘adhocTP‘ exprFail
```

The use of failTP as the default strategy is required, as innermost reaches the fixed-point when *step* fails. It should be noticed that if we use idTP instead, *step* always succeeds, resulting in an infinite loop of this strategy!

As a consequence, we have also to update the worker function so that the default behaviour does not return a tree, but returns *Nothing* instead.

```
exprFail :: Exp → Maybe Exp
exprFail (Add e (Const 0))          = Just e              -- (1)
exprFail (Add (Const 0) t)          = Just t              -- (2)
exprFail (Add (Const a) (Const b)) = Just (Const (a + b))  -- (3)
exprFail (Sub a b)                  = Just (Add a (Neg b))  -- (4)
exprFail (Neg (Neg f))              = Just f              -- (5)
exprFail (Neg (Const n))            = Just (Const (−n))    -- (6)
exprFail _                          = Nothing             -- default
```

If we run *optimizeIM* with *letExample*, then we get the desired result:

```
letResult = let a =  b + 16
                c =  8
                w = let y = let q = 7
                            in  q
                        z =   a
                    in z + b
                b =  (c + 3) − c
            in c + a − w
```

Once Top Down Strategy: This strategy traverses the tree in a top down traversal scheme and it stops as soon it is able to apply the worker function. Thus, it applies the type specific transformation only once.

Next we show the the optimization performed using the *oce_tdTP* strategy, which is very similar to the full_tdTP solution we presented before.

```
optimizationOTD :: Zipper Let → Maybe (Zipper Let)
optimizationOTD t = applyTP (once_tdTP step) t
  where step = idTP `adhocTP` expr
```

When we apply this strategic function to our *letExample* we get exactly the same tree/input! In fact, because the default behaviour of our *step* function is to apply idTP, which succeeds for any node, we are able to perform "work" in the first reached node, therefore stopping the strategy immediately. Even if the first visited node matches type with *expr* and it is applied, since it always returns a tree (even if it is the same, unchanged tree), the once_tdTP strategy immediately applies it once and stops.

To use this strategy we need to have a worker function that fails as the default behavior. Thus, the next and correct definition uses the *wokerFail* transformation and the failTP default case.

```
optimizationOTD  :: Zipper Let → Maybe (Zipper Let)
optimizationOTD t = applyTP (once_tdTP step) t
  where step = failTP `adhocTP` workerFail
```

As defined by this strategy, if we apply *optimizationOTD* to the **let** example, there is only one optimization performed: the addition of 3 and 4 in the definition of q, which is the first subtree reached in a top down recursion scheme.

```
letResult = let a  = b + 16
                c  = 8
                w = let y = let q = 7
                            in q
                        z = 0 + a + 0
                    in z + b
                b = (c + 3) − c
            in c + a − w
```

Once Bottom Up Strategy: This strategy traverses the tree in a bottom up traversal scheme and it stops as soon as it is able to apply the worker function. Similarly to its top down counterpart, this strategy applies the type specific transformation only once. This strategy also requires that the worker function fails in its default rule, so that the exact same input tree is not produced as result.

When we use this strategy once_buTP to apply the worker function *workerFail* to our **let** example, the first optimization that is able to perform in the addition with 0.

```
letResult = let a  = b + 16
                c  = 8
```

```
                       w = let y = let q = 3 + 4
                                   in q
                               z = a + 0
                           in z + b
                       b = (c + 3) − c
                   in c + a − w
```

Stop Top Down Strategy: This strategy applies a strategy while traversing the tree top down. However, it does not perform a full top down traversal: when the worker function is applied, then it does not go deeper in that subtree. The strategy continues performing a top down traversal in the rest of the tree.

```
optimizationSTD :: Zipper Let → Maybe (Zipper Let)
optimizationSTD t = applyTP (stop_tdTP step) t
  where step = failTP `adhocTP` workerFail
```

```
letResult = let a  = b + 16
                c  = 8
                w = let y = let q = 7
                            in q
                        z = 0 + a
                    in z + b
                b = (c + 3) − c
            in c + a − w
```

However, if we consider the following input **let** example, where we switched the two definitions in the nested let.

```
example = let a = b − 16
              c = 8
              w = let   z = 0 + a + 0
                        y = let q = 3 + 4
                            in q
                    in z + b
              b = (c + 3) − c
          in c + a − w
```

then we get

```
letResult = let a  = b + 16
                c  = 8
                w = let z = 0 + a
                        y = let q = 3 + 4
                            in q
                    in z + b
                b = (c + 3) − c
            in c + a − w
```

4.1 Type Unifying Strategies

Next, we show an example using a type-unifying strategy. We define a function *names* that collects all defined names in a **let** expression. First, we define a function *select* that focuses on the *Let* tree nodes where names are defined, namely, *Assign* and *NestedLet*. This function returns a singleton list (with the defined name) when applied to these nodes, and an empty list in the other cases.

```
select :: List → [Name]
select (Assign s _ _)    = [s]
select (NestedLet s _ _) = [s]
select _                 = []
```

Now, *names* is a Type-Unifying function that traverses a given *Let* tree (inside a zipper, in our case), and produces a list with the declared names.

```
names  :: Let → [Name]
names t = applyTU (full_tdTU step) (toZipper t)
          where step = failTU `adhocTU` select
```

The traversal strategy influences the order of the names in the resulting list. We use a top-down traversal so that the list result follows the order of the input. This is to say that

```
names (toZipper example) = ["a", "c", "w", "y", "q", "z", "b"]
```

If we use a full bottom-up strategy, it produces the reverse of this list.

To use the once/stop down/Bottom up strategies we need to change the worker function so that it reports failure in the default alternative. Yet again, we return a *Maybe* result, using type constructor *Nothing* to report failure.

```
select :: List → Maybe [Name]
select (Assign s _ _)    = Just [s]
select (NestedLet s _ _) = Just [s]
select _                 = Nothing
```

now, we can write

```
namesOBUF :: Let → [Name]
namesOBUF t = fromJust $ applyTU (once_buTU step) (toZipper t)
  where step = failTU `adhocTU` selectF
```

that given *example* as input, it produces the singleton list ["b"].

4.2 Refactoring Haskell Programs

Source code smells make code harder to comprehend and they do occur in any programming language. *Haskell* is no exception. For example, inexperienced *Haskell* programmers often write $l \equiv []$ that uses equality to check whether a list is empty, instead of using the predefined *null* function. Within a strategic programming style we express the elimination of this known smell by defining the type-specific transformation which is defined in two simple alternatives: A first one, specifies the pattern of $l \equiv [\,]$ and the transformation to be performed. A second one defines the work to be performed in all other parts of *Haskell* programs/trees: nothing in this case. In the first alternative, the original and refactored patterns are expressed using the data type defined the full abstract syntax of *Haskell*, as defined in its libraries. By just knowing the types and constructors that represent both $l \equiv [\,]$ and *null l*, we write the following type-specific function:

```
nullList :: HsExp → Maybe HsExp
nullList (HsInfixApp a (HsQVarOp (UnQual (HsSymbol "=="))) (HsList []))
            = Just (HsApp (HsVar (UnQual (HsIdent "null"))) a)
nullList _ = Nothing
```

Now, we define a strategic-based function that applies the *nullList* transformation to a full *Haskell* abstract syntax tree. A refactoring is a type preserving transformation, and in this case we rely in a full bottom-up tree traversal.

```
smells :: Zipper HsModule → Maybe (Zipper HsModule)
smells h = applyTP (full_buTP worker) h
      where worker = failTP ‘adhocTP‘ nullList
```

This is the full *Haskell* program that eliminates this known smell. Together with the front-end included in the *Haskell* libraries, we produced an useful refactoring tool: for example, it processed 1139 *Haskell* files (totaling 82124 lines of code), written by first year students, and eliminated 850 code smells in those files [9].

References

1. S. P. Luttik, E. Visser, Specification of rewriting strategies, in: Proceedings of the 2nd International Conference on Theory and Practice of Algebraic Specifications, Algebraic'97, BCS Learning & Development Ltd., Swindon, GBR, 1997, p. 9.
2. E. Visser, Z.-e.-A. Benaissa, A. Tolmach, Building program optimizers with rewriting strategies, in: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, ICFP '98, Association for Computing Machinery, New York, NY, USA, 1998, p. 13–26.
URL `https://doi.org/10.1145/289423.289425`
3. G. Huet, The Zipper, Journal of Functional Programming 7 (5) (1997) 549–554.
URL `https://doi.org/10.1017/S0956796897002864`
4. M. D. Adams, Scrap your zippers: A generic zipper for heterogeneous types, in: Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming, WGP '10, New York, NY, USA, 2010, pp. 13–24.
URL `https://doi.org/10.1145/1863495.1863499`
5. R. Lämmel, S. P. Jones, Scrap your boilerplate: A practical design pattern for generic programming, in: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, TLDI '03, New York, NY, USA, 2003, pp. 26–37.
URL `https://doi.org/10.1145/640136.604179`

6. R. Lämmel, J. Visser, Typed combinators for generic traversal, in: S. Krishnamurthi, C. R. Ramakrishnan (Eds.), Practical Aspects of Declarative Languages, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 137–154.
URL `https://doi.org/10.1007/3-540-45587-6_10`
7. R. Lämmel, J. Visser, A Strafunski Application Letter, in: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages, PADL '03, Springer-Verlag, Berlin, Heidelberg, 2003, p. 357–375.
URL `https://doi.org/10.1007/3-540-36388-2_24`
8. L. Kramer, E. Van Wyk, Strategic tree rewriting in attribute grammars, in: Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 210–229.
URL `https://doi.org/10.1145/3426425.3426943`
9. J. N. Macedo, M. Viera, J. Saraiva, Zipping strategies and attribute grammars, in: Functional and Logic Programming: 16th International Symposium, FLOPS 2022, Kyoto, Japan, May 10–12, 2022, Proceedings, Springer-Verlag, Berlin, Heidelberg, 2022, p. 112–132.
URL `https://doi.org/10.1007/978-3-030-99461-7_7`