A Generic Recursion Toolbox for Haskell

Or: Scrap Your Boilerplate Systematically

Deling Ren   Martin Erwig

School of EECS
Oregon State University
[rende, erwig]@eecs.oregonstate.edu

Abstract

Haskell programmers who deal with complex data types often need to apply functions to specific nodes deeply nested inside of terms. Typically, implementations for those applications require so-called boilerplate code, which recursively visits the nodes and carries the functions to the places where they need to be applied. The scrap-your-boilerplate approach proposed by Lämmel and Peyton Jones tries to solve this problem by defining a general traversal design pattern that performs the traversal automatically so that the programmers can focus on the code that performs the actual transformation.

In practice we often encounter applications that require variations of the recursion schema and call for more sophisticated generic traversals. Defining such traversals from scratch requires a profound understanding of the underlying mechanism and is everything but trivial.

In this paper we analyze the problem domain of recursive traversal strategies, by integrating and extending previous approaches. We then extend the scrap-your-boilerplate approach by rich traversal strategies and by a combination of transformations and accumulations, which leads to a comprehensive recursive traversal library in a statically typed framework.

We define a two-layer library targeted at general programmers and programmers with knowledge in traversal strategies. The high-level interface defines a universal combinator that can be customized to different one-pass traversal strategies with different coverage and different traversal order. The lower-layer interface provides a set of primitives that can be used for defining more sophisticated traversal strategies such as fixpoint traversals. The interface is simple and succinct. Like the original scrap-your-boilerplate approach, it makes use of rank-2 polymorphism and functional dependencies, implemented in GHC.

Categories and Subject Descriptors D.1.m [*Programming Techniques*]: Generic Programming; D.2.13 [*Software Engineering*]: Reusable Libraries

General Terms Design, Languages

Keywords Generic Programming, Traversal Strategy

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Haskell'06 September 17, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-489-8/06/0009. . . $5.00.

1. Introduction

Lämmel and Peyton Jones address the problem of traversing recursive data structures in their papers [12, 13, 14]. They propose a design pattern to eliminate boilerplate code by applying a generic programming technique. In the following we briefly summarize some major elements of their approach.

The examples given in [12] are based on a collection of data types that represent a simplified structure of a company. We repeat the definitions here for reference:

```
data Company  = C [Dept]
data Dept     = D Name Manager [Unt]
data Unt      = PU Employee | DU Dept
data Employee = E Person Salary
data Person   = P Name Address
data Salary   = S Float
type Manager  = Employee
type Name     = String
type Address  = String
```

Derived instances for type classes `Typeable` and `Data` are declared but omitted here for clarity. Here is an example definition of a company according to the above data types:

```
genCom :: Company
genCom = C [D "Research" joe [PU mike, PU kate],
            D "Strategy" mary []]

joe, mike, kate, mary :: Employee
joe  = E (P "Joe"  "Oregon") (S 8000)
mike = E (P "Mike" "Boston") (S 1000)
kate = E (P "Kate" "San Diego") (S 2000)
mary = E (P "Mary" "Washington") (S 100000)
```

A simple transformation task is to define an `increase` function that increases everybody's salary by a certain percentage. Normally, in Haskell we would have to define one increase function for each individual data type. The only purpose these functions serve is to traverse the data types and move the `incS` function to the `Salary` type where it actually increases the salary:

```
incS :: Float -> Salary -> Salary
incS k (S s) = S (s * (1+k))
```

This `incS` function is the only interesting bit. All other code is "boilerplate code". As the sizes of the data types grow, the boilerplate code becomes extremely clumsy and hard to maintain. It also does not scale up well. Changes to the data type definitions will entail many changes in the boilerplate codes. In [12], a type extending function `mkT` is introduced that, when applied to functions like `incS`, produces a generic transformation. A generic transformation is polymorphic. When applied to a `Salary`, it behaves the same as `incS`, otherwise it behaves like the identity function. The

type extension is implemented with a `cast` function, which is a member function of `Typeable` type class. The `cast` function performs a safe type casting. Type class `Data` extends `Typeable` and has a member function `gfoldl`, which will be discussed in Section 3.3, that is essential to defining one-layer traversals.

A generic traversal combinator `everywhere` is also provided that traverses a term recursively and applies a generic transformation to every node in the term.

With the generic traversal combinator, programmers only need to implement the interesting part of recursive traversals, the `incS` function, and feed them to `mkT` and `everywhere` to achieve the same goal as the boilerplate code. The definition of the `increase` function defined in [12] is repeated here for reference:

```
increase :: Float -> Company -> Company
increase k = everywhere (mkT (incS k))
```

The scrap-your-boilerplate (SYB) approach relieves a big burden from Haskell programmers who need to traverse complex data structures frequently. They can now focus on the code that does the real job instead of the traversal itself. The boilerplate code to traverse arbitrary data structures can be automatically derived. In the following, we illustrate how to implement some traversals in our library through several examples. We begin with defining the `increase` function using our interface:

```
increase :: Float -> Company -> Maybe Company
increase k = traverse Trans NoCtx Full FromBottom FromLeft
                      (always (incS k))
```

Compared to the original version, this `increase` function is defined using more parameters which specify the traversal. In this particular case, the parameters define the traversal to be a transformer that modifies nodes, independently of contextual information. It is a full traversal (all nodes in the tree will be visited), and the order of visiting the nodes is from bottom to top, from left to right. Another noticeable difference is the return type, which is a `Maybe` accounting for possible failures. We allow a transformation on a node to fail. A failed transformation will leave the node unchanged. Such mechanism can be used to construct contingent transformations. We will discuss more about failures in Section 2. The "interesting case" that deals with `Salary` data is still the `incS` function, which we can reuse without changes. However, instead of extending its type to make it a generic function, we take a slightly different approach. We define a few combinators to combine specific functions and pass a list of them to the traversal combinator. In this case, the combinator `always` takes the specific function `incS k`. This specific function is unconditionally applied and works on any term of type `Salary`.

In applications like this one, not all the parameters are interesting. The users usually do not care, or even do not know, about the context and the left-to-right traversal direction. All they need is a transformation. We have identified default values for the different dimensions along which a traversal can be customized and have introduced functions for all possible combinations of parameters following a strict naming scheme that will be explained in detail in Section 3.3. Employing the traversal that represents the shown traversal parameters, the presented example can be defined much more succinctly as follows.

```
increase :: Float -> Company -> Company
increase k = transformB (always (incS k))
```

The `B` indicates "bottom-up", which was chosen in the original SYB approach. The top-down version `transform` works just as well.

In the following we continue to use the expanded versions of the traversals to make the parameters and options explicit.

Our next example is an accumulation instead of a transformation. An accumulation can serve as a query defined in [12] but is more general. The following function computes the salary bill for a company by traversing the company data structure and accumulates all salaries.

```
bill :: Company -> Maybe Float
bill = traverse Accum NoCtx Full FromTop FromLeft
                (always col) 0
    where col a (S s) = a + s
```

The local function `col` takes an accumulator, which is the sum collected so far, and a `Salary` and adds the salary to the sum. In the end of the traversal, the accumulator is the sum of all salaries.

1.1 Possible Extensions

1.1.1 Accumulation and Transformation

Suppose we not only want to increase everyone's salary, but also need the total amount being increased. We keep traversing the company data structure, increasing everyone's salary and modify the total amount at the same time. Again, we need to resort to a combinator that is similar to `everywhere`, but can maintain a state for the total. Such a function can be defined as follows. Upon a successful return, the result consists a total amount and a modified company value.

```
incBill :: Float -> Company -> Maybe (Float,Company)
incBill k = traverse AccTrans NoCtx Full FromTop FromLeft
                     (always (colS k)) 0

colS :: Float -> Float -> Salary -> (Float, Salary)
colS k a (S s) = (a+k*s, S (s*(1+k)))
```

A similar application in program transformation occurs when we need to generate new variables that do not conflict with any existing variables in the original program. We need to keep track of variables that have been already generated to keep the variable names unique. In general, a transformation might need to access information accumulated from the nodes visited so far in the traversal. Accumulations find a broad range of applications in language processing area. Examples include counting certain nodes, collecting variables, collecting other constructs, etc.

1.1.2 Partial Traversals

In some applications, not all the nodes in a term have to be visited. Consider a local transformation where we only want to apply the transformation to a certain part of the term. One such application is increasing salaries in a certain department rather than the whole company. This problem is addressed in Section 6.2 in [12] with a function `incrOne` defined using the `gmapT` function, which is rather complicated to come up with for ordinary programmers. Since a similar pattern can be observed in many applications, it would be beneficial to provide a general solution once and for all. An elegant way to realize such a transformation is to employ a so-called *stop-traversal* [11]. A stop-traversal tries to apply a visit to all nodes. If the visit succeeds on a node, the traversal continues without descending into that node. In this example, another traversal is passed as a visit argument to the outer traversal. The nested traversal is the `increase` function. It is applied to nodes that are departments with a matching name. The `mwhenever` function is used to construct a conditional visit and will be explained in Section 3.1.

```
incOne :: Float -> Name -> Company -> Company
incOne k d = traverse Trans NoCtx Stop FromTop FromLeft
                      (increase k `mwhenever` isDpt d)

isDpt :: Name -> Dept -> Bool
isDpt d (D n _ _) = n==d
```

We can also consider *once-traversals* [25] where we only want to apply a transformation once. These are also a special case of partial traversals. For instance, we can increase the first salary we encounter when traversing the company data.

```
incFst :: Float -> Company -> Maybe Company
incFst k = traverse Trans NoCtx Once FromTop FromLeft
                    (always (incS k))
```

1.1.3 Traversal with Contexts

Transformations that depend on non-local data are also difficult to express in the original SYB approach. Let us consider a more complicated application of increasing salaries. Say we want to adjust the increase rate according to the department. A context, which is the increase rate, is carried through the traversal. It is initialized to a default rate and is updated whenever the traversal is descended into a node so that all salaries inside that node will get increased by the new rate (unless the rate gets changed again before that salary is reached).

```
incDpt :: Float -> Company -> Maybe Company
incDpt = traverse Trans Ctx Full FromTop FromLeft
                  (mk (\c d -> lookupRate d))
                  (always incS)
```

Compared to the previous examples, this contextual traversal takes as an additional argument a context updater (`\c d -> lookupRate d`), where the function `lookupRate` determines the increase rate for the department. Similar to visits, a context updater will be applied to terms of any type. Therefore, it needs to be generic as well. The `mk` function is used to wrap a specific context updater and make it generic. It will be explained in Section 3.1 along with combinators for visits.

The careful reader might have noticed that the visit in this example, expressed by `always incS`, has a different type than before. Here, `incS` is used as a *contextual visit*, which takes an extra parameter, the context. The types of all 6 visits are listed in Table 2. The `always` function is overloaded in order to provide a uniform interface to the programmer.

We can also consider an application in language processing. Suppose we want to implement a beta reduction for lambda calculus. A beta redex is a lambda abstraction applied to an argument. The body of the lambda abstraction is traversed so that all the free occurrences of the bound variable are replaced by the argument. However, we have to be careful not to replace locally bound variables with the same name. When we descend into the term, we need to keep track of a collection of bound variables. The transformation needs to check against these variables.

From these two problems, we can generalize a pattern of contextual traversal. An initial context is passed to the traversal and it gets updated by an update function when descending into subterms. A beta reduction carries a list of bound variables as the context and it gets extended at lambda abstractions.

1.2 Contribution and Organization of This Paper

The shown applications can be generally implemented by employing the generic fold operator `gfoldl` defined in [12]. However, this is not at all a trivial task. Our goal is to generalize the design pattern and extend it to support these applications. The approach we take is to combine contributions from SYB, Strafunski, and Stratego and to create a fully typed generic traversal library consisting of categorized recursive traversal strategies and implement the library with strategy combinators.

Lämmel and Visser present a combinator library for generic traversals and a set of traversal schemes as part of Strafunski [15, 16, 11]. However, it relies on DrIFT to generate the instances of type class `Term`. Also, it is not a fully statically-typed approach.

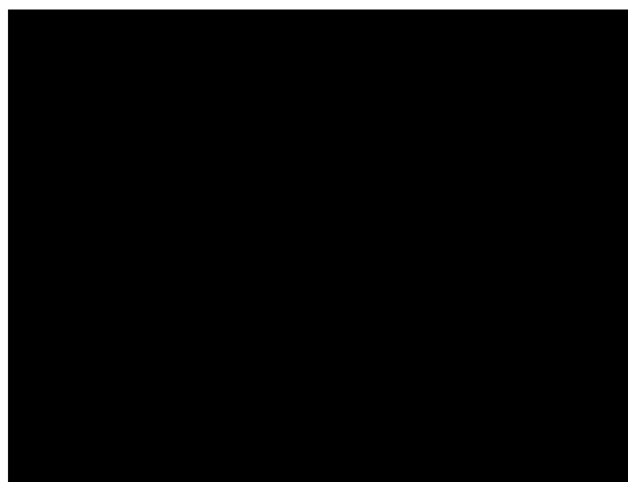

Figure 1. Haskell Recursion Library integrating ideas and concepts from Stratego, Strafunski, and SYB

One uses an abstract datatype for generic functions to separate typed and untyped code. In [11], Lämmel presents a hierarchy of traversals and defines a *traverse* function that can be highly parameterized. We make use of this "mother of traversals" to derive all traversals.

Stratego [25, 24, 2] defines an abundant set of traversal strategies. Our main motivation comes from the need to apply these traversal strategies in our program transformation tool [7]. However, we want to use them in the context of Haskell. We also want the static type safety, which is not found in Strafunski and Stratego. We are also motivated by the need for a concise program interface without using complex data types such as monads. Therefore, we propose the approach of defining a generic traversal library with a simple and general programming interface and a rich set of traversal strategies. In this experimental implementation, we focus on the concepts rather than having a complete set of traversal strategies. However, with the genericity of the approach, new traversal strategies can be defined easily. The relationship between this library and Stratego, Stranfunski, and SYB is sketched in Figure 1. The source code of the library can be obtained online [18]. Features of different approaches are compared in Table 1.

	Reclib	Stratego	Strafunski	SYB
Typed	✓			✓
Integrated in Haskell	✓		✓	✓
Strategies	✓	✓	✓	
Concrete Syntax		✓		

Table 1. Feature comparison of Haskell Recursion Library, Stratego, Strafunski, and SYB.

In the rest of this paper, we categorize in Section 2 the problem domain of traversals by extracting five parameters that are, to a large extent, orthogonal to each other. In Section 3, we describe a high-level programming interface. This interface provides a means to parameterize traversal strategies that cover all possible combinations of those five parameters. In the core of the interface, we define one generic traversal strategy that is the "mother" of all one-pass traversal strategies we explored. An intermediate layer of programming interface is also defined for users who require more than one-layer traversals. This interface is concise and clean. Two

fixpoint strategies, innermost and outermost, are studied and implemented using the interface as examples for extendibility. Subsections 3.4 3.5, and 3.6 elaborate implementation details and can be skipped without jeopardizing understanding of the programming interface. In Section 4 we illustrate more examples that make use of the library in greater detail. In Section 5, we present a practical application of our library. We discuss and compare related work in Section 6. In Section 7 we present conclusions and directions for future work.

2. Design Space

In a typical traversal, all or part of the nodes are visited in a particular order. We use the term *visit* to refer to one access to a particular node. During a visit, information is retrieved from the node, and/or the node is modified. The information and the modification might depend on the information retrieved from the nodes already visited in the traversal and/or the path from the node to the root.

A visit that retrieves information does so by taking already accumulated information and returning the new accumulator, which is threaded through all the visits in the traversal. We distinguish three kinds of visits. A *transformer* modifies a node without retrieving information, an *accumulator* retrieves information without modifying the node, and an *accumulating transformer* does both simultaneously. We borrow this categorization from [20]. Every visit may either succeed or fail. Therefore, the result of a visit is wrapped in a `Maybe` data type.

In the example of increasing salaries for people in a certain department, the traversal combinator needs to carry some information related to the path from the current node to the root. We call this a *context*. The combinator updates the context by applying a user-provided *context function*. It then passes the updated context to all the children of the node.

Therefore, there are all together six kinds of visits whose types are listed in Table 2. As a convention, we use `c` to denote a context type, `a` for an accumulator type and `t` for a term type.

	Contextual	Non-contextual
1	`c -> t -> Maybe t`	`t -> Maybe t`
2	`c -> a -> t -> Maybe a`	`a -> t -> Maybe a`
3	`c -> a -> t -> Maybe (a,t)`	`a -> t -> Maybe (a,t)`

1. Transformation 2. Accumulation 3. Accumulating transformation

Table 2. Types of 6 Kinds of Visits

A traversal can be categorized regarding the number of times every node is visited. A *one-pass traversal* traverses a tree in one pass and visits each node at most once. A typical example is a depth-first search. A *fixpoint traversal* [2] applies a visit to a tree using a certain strategy repeatedly until it is not applicable anymore. Innermost and outermost traversals fall into this category. We implement both kinds of traversals in our library, but we focus more on one-pass traversals.

In a one-pass traversal, it is not always desirable to visit all the nodes in a term. A typical scenario of a partial traversal is when we abort the traversal after a single successful visit. This kind of *coverage* is called *once* as opposed to *full* where all nodes are visited sequentially unless stopped by a failed visit. Another common situation is a so-called *stop*-traversal that tries to apply a visit to the root node of a tree. If it fails, it then tries to recursively apply it to all children. Otherwise, it stops. Effectively, a stop-traversal visits nodes on a frontier of a tree. A typical application for such traversals is optimization. We can significantly decrease the number of nodes visited by focusing on interesting nodes. Symmetrically, a *spine*-traversal visits a chain of nodes from the root to a leaf. A spine-traversal fails if no spine exists such that the visit succeeds on every node on the spine. Figure 2 illustrates these four different kinds of coverage. In the figure, the dashed line connects all nodes that are successfully visited, but does not include those tried but failed.

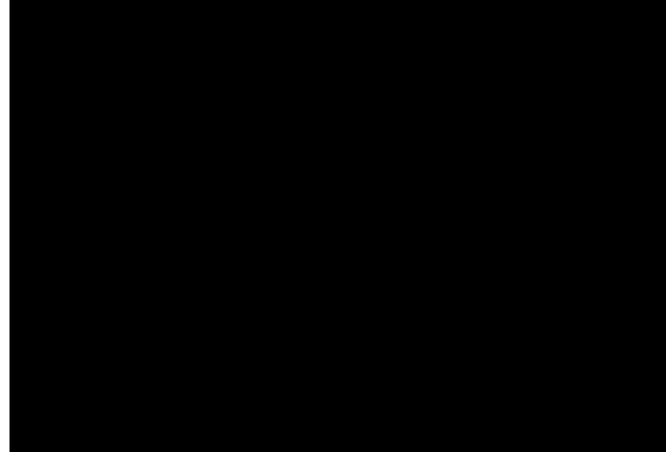

Figure 2. Traversals of 4 Kinds of Coverage

Furthermore, there are two kinds of directions that affect the order in which the nodes are visited: the *vertical direction* and the *horizontal direction*. A vertical direction can be either *top-down* or *bottom-up*. A horizontal direction is either *left-to-right* or *right-to-left*. In a top-down traversal, a root is visited before its descendants. In a bottom-up traversal, the children are visited before their parent. Top-down and bottom-up traversals are also often referred to as preorder and postorder traversals, respectively. The order in which the siblings of a common parent are visited is determined by the horizontal direction, which can be either from the left or from the right. The directions usually matter for the accumulating traversals or once-traversals. Figures 3 and 4 illustrate vertical and horizontal directions, respectively.

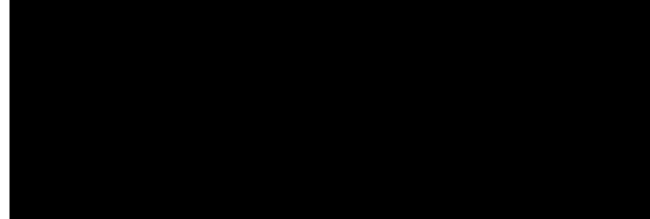

Figure 3. Vertical Direction

In summary, five parameters determine a (one-pass) traversal: kind of the visit, context, coverage, and two directions. These parameters are mostly orthogonal to each other. We can obtain a rich set of traversals by customizing all these parameters.

3. Programming Interface

Our goal is to provide an easy-to-use and effective programming interface to users who wish to program generic traversals. In this

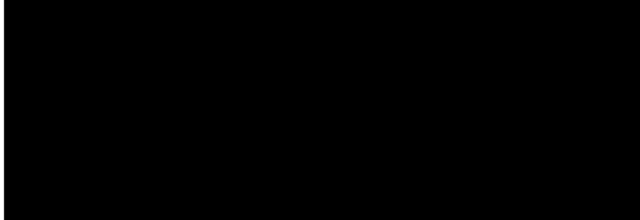

Figure 4. Horizontal Direction

section, we will describe the generic traversal combinators and some necessary helper functions. The interface is divided into two layers. A higher-level interface is provided to users who do not have profound knowledge in generic programming and term traversals. They can easily program their own traversals using provided combinators and compose necessary arguments using the auxiliary functions. The library is flexible and extensible in the sense that an intermediate layer is exposed to users who wish to write traversal strategies that are not found in our library to meet their own needs. As an example, the implementation of fixpoint traversal combinators `innermost` and `outermost` will be presented. Other example traversals include *downup* and *updown* strategies [23, 2]. They, too, can be implemented with the intermediate layer of our library.

3.1 Building Generic Functions

The visits as well as context updaters are generic in the sense that they are applicable to values of any type. The `mkT` function described in [12] creates a generic function of type `a -> a` out of a specific function, and `extT` extends a generic function with a specific function. However, this approach of using two combinators does not work for our purpose for two reasons. First, our visits return `Maybe` values. Second, we cannot expect one set of combinators to work for all kinds of visits, because they generally have different types, as we have seen in Table 2. Having a separate set of such `mkT` and `extT` combinators for each kind of visit is very cumbersome. Therefore we decided to provide a universal mechanism for composing visits and hide the differences and details. The decision resulted in the design that the generic traversal function `traverse` (which will be explained in Section 3.2) takes a list of specific visits (and possibly context updaters) rather than a generic one. This also relieves the users of the burden of applying the extending combinator. We need to encapsulate specific context updaters and visits with rank-2 polymorphic data types so that they can be put into lists. An example of such a data type for a contextual accumulating transformer is the data type `GenCAT`, defined as follows.

```
data GenCAT c a = forall t. Typeable t =>
                  GenCAT (c -> a -> t -> Maybe (a,t))
```

Specific visit functions that work on different types of nodes (but on the same context and accumulator types) can be wrapped with the data constructor `GenCAT` and put in a list which is passed to the `traverse` function. For each kind of visit listed in Table 3, a separate data type is required. The context updater works in a similar way.

```
data GenU c a = forall t. Typeable t =>
       GenU (c -> t -> c)
```

To hide the differences between these data constructors, one overloaded function `mke` is provided. It serves a similar purpose as `mkT` function except that it works for all visits and context updaters. A specific visit or context updater is passed to the `mke` function, and a generic one is constructed. For contextual accumulating transformers, its type is the following.

```
mke :: (c -> a -> t -> Maybe (a,t)) -> GenCAT c a
```

Since in many cases a generic function is built from just one specific function, a function `mk` is defined to further hide the list structure.

```
mk x = [mke x]
```

In fact, even if two or more specific functions are used to compose a generic one, the `mk` function can be used, and the results can be concatenated using `++` operator. Therefore clients usually do not need the `mke` function.

In addition to the `mk` combinator, we provide two sets of combinators for composing visits. To selectively apply one of two visits depending on the node, a combinator `mcond` is provided, which takes one predicate and two visits. It implements a conditional. In cases where the else part is missing (indicating a failed visit), the combinator `mwhenever` can be used. To apply a visit unconditionally, the combinator `malways` is used. A visit returns a `Maybe` value to indicate a success or failure. For visits that do not fail, it is an extra burden to handle the `Maybe` data type. We define three symmetric combinators `cond`, `whenever`, and `always` that take visits that do not return `Maybe` values. In the salary-increasing example, the visit can be composed using `always`: `always (incS k)`. A visit that increases every salary inside a node if the node is a certain department can be composed with the `increase` function and a predicate, that takes the department name as a parameter `d`.

```
increase k `mwhenever` \(D n _ _) -> n==d
```

Such a visit can be used to compose a stop-traversal. It is recursively tried on every node in a term but has no effect on the node unless it is a department whose name matches `d`, in which case the `increase` function is applied recursively to the subtrees of that node.

3.2 Traversal Engine

The main component of the interface is a heavily overloaded function `traverse` that can be customized by all the five parameters we mentioned. And since it is an overloaded polymorphic function, its type varies. It is defined as a member function of type class `Traversal`:

```
class Traversal u v c a t x | u v c a t -> x  where
      traverse :: u -> v -> Coverage -> VD -> HD-> x
```

What is common to all instances are the first five parameters that identify a traversal. Type variable `u` represents the kind of visit, and `v` is either `Ctx` or `NoCtx` representing the presence or absence of the context. As explained in Section 2, type variables `a`, `c`, and `t` represent the types of the accumulator, context and term, respectively. The functional dependency helps the type system determine the instance of `traverse` when it is applied but the result type is not explicitly specified. The reader might wonder whether `c`, `a`, and `t` are really needed since they do not appear in the type of `traverse`. The answer is yes, they are indeed required, because `x`, the type of the traversal, depends on them.

Presented below are the data type definitions for the parameters of `traverse`..

	v = Ctx	v = NoCtx
1	[GenU c a] -> [GenCT c a] -> c -> t -> Maybe t	[GenT c a] -> t -> Maybe t
2	[GenU c a] -> [GenCA c a] -> c -> a -> t -> Maybe a	[GenA c a] -> a -> t -> Maybe a
3	[GenU c a] -> [GenCAT c a] -> c -> a -> t -> Maybe (a,t)	[GenAT c a] -> a -> t -> Maybe (a,t)

1. u = Trans 2. u = Accum 3. u = AccTrans

Table 3. Types of Traversals

```
data Trans = Trans
data Accum = Accum
data AccTrans = AccTrans

data Ctx = Ctx
data NoCtx = NoCtx

data Coverage = Full | Spine | Once | Stop
data VD = FromTop | FromBottom
data HD = FromLeft | FromRight
```

Kind of visit and context presence are defined using one data type for each kind as opposed to the other three parameters in which each kind is represented by just one data constructor. This is simply a means for the compiler to choose the correct instance of `traverse` function. The rest of the parameters and the result type are all combined in `x`, which is the traversal type, determined by `u`, `v`, and the types of the accumulator, the context and the term. For example, an instance of contextual accumulating transformers takes the following form.

```
instance Data t => Traversal AccTrans Ctx c a t
                     ([GenU c a] -> [GenCAT c a]
                      -> c -> a -> t -> Maybe (a,t))
  where ...
```

A complete list of correspondence between `x`, `u`, and `v` is listed in Table 3.

A list of context updaters (`[GenU c a]`) has to be provided for contextual traversals. A list of visit functions is required for all traversals. The type of the visit depends on the kind of the visit and presence of context. The most general visit, a contextual accumulating transformer, has the following type, defined as a type synonym (`GCAT` is not an abstract data type and should not be confused with `GenCAT` previously mentioned).

```
type GCAT c a = forall t. Data t =>
                c -> a -> t -> Maybe (a, t)
```

where `c` is the type of the context, `a` is the type of the accumulator, and `t` is a universally quantified type variable, which means that a visit is a rank-2 polymorphic function that should be applicable to values of any type. We provide auxiliary combinators for composing such generic functions out of specific functions as we have seen in Section 3.1. The result type of this visit, `Maybe (a,t)`, captures the nature of an accumulating transformer. Upon success, a new accumulator and a modified node are returned. The visit returns `Nothing` to signal a failure. The action to be taken upon a failed visit depends on the traversal: Full-traversal or spine-traversal fail immediately, whereas once- and stop-traversals continue. However, while a once-traversal continues with the subterms only until a successful visit, a stop-traversal continues even after a successful visit, it only stops descending into subterms. The types of other kinds of traversal can be deduced naturally. For non-contextual visits, the `c` is omitted, transformers will not have the `a`, and an accumulator returns a value of type `Maybe a` instead.

3.3 Syntactic Sugar

The `traverse` function is the ultimate interface for the programmers. However, programmers are not always interested in all the traversal parameters. In the example of increasing everyone's salary, the traversal order has no effect on the result. For cases like this, we define instances of the `traverse` function using default values. We introduce 96 functions, each of which is a partial application of `traverse` function to a combination of the traversal parameters. The functions follow a naming convention. The name consists of a *verb* and an optional prefix and three optional suffixes. The verb is either `transform`, `accumulate`, or `acctrans`. The prefix specifies the coverage, which defaults to full, when omitted. The first suffix is the presence of the context. A letter `C` follows the verb to obtain a contextual traversal, an absence indicates a non-contextual traversal. What follows is the vertical direction. A letter `B` indicates a bottom-up traversal. When it is omitted, a top-down traversal is obtained. Finally, a `'` symbol can be appended to the end to obtain a right-to-left traversal instead of the default left-to-right version.

According to these naming rules, a contextual, bottom-up, right-to-left accumulation corresponds to function `accumulateCB'` of the following type.

```
Data t => [GenU c a] -> [GenCA c a] ->
          c -> a -> t -> Maybe a
```

With the conventions, the functions defined in Section 1.1 can be given in a more succinct way:

```
increase k =      transformB (always (incS k))
bill       =      accumulate (always col) 0
incBill k  =      acctrans   (always (colS k)) 0
incOne k d = stopTransform  (increase k `mwhenever` isDpt d)
incFst k   = onceTransform  (always (incS k))
incDpt k d =      transformC (mk (\c d -> lookupRate d))
                             (always incS))
```

3.4 Crafting Traversals

The combinators we presented above provide enough flexibility for defining commonly used one-pass traversals. But more complicated traversals, such as a fixpoint traversal *innermost* which might visit some nodes more than once, cannot be expressed. To help users who have knowledge in traversal strategies and need to define special traversals, the library also exposes an intermediate layer. In the rest of this section we explain how the recursive traversal strategies are defined using the intermediate layer.

A basic component of every traversal strategy is a one-layer strategy. Such a strategy does not apply a visit recursively. Instead, it applies another strategy to the immediate subterms. We define four such combinators. Strategy `all_l` applies a strategy to all the immediate subterms of a node in a left-to-right order. Strategy `one_l` tries a strategies on all subterms of a term and stops after a successful application. The other two, `all_r` and `one_r`, are their right-to-left counterparts. Recursive traversals can then be built on these one-layer strategies. For instance, a top-down full-traversal can be conceptually defined as follows.[1]

$$\mathsf{fulltd}(v) \quad = \quad v; \mathsf{all}(\mathsf{fulltd}(v))$$

where v is the visit to be applied. The sequential composition operator ; [24] takes two strategies and applies them sequentially. Failure of either one will cause the failure of the whole strategy. Instantiating all [24] in the above definition with `all_l` and `all_r` will result in left-to-right and right-to-left versions of top-down full-traversals. A one-layer strategy does not need to take into

[1] The definition is taken from that of the topdown strategy in [25], but renamed here for the naming consistence.

consideration the context because all immediate subterms will have the same context. It is the job of the recursive traversal strategies to update the context and pass it to one-layer traversals. We define a type synonym for a one-layer traversal without a context:

```
type GAT a = forall t. Data t => a -> t -> Maybe (a,t)
```

It is a generic function that takes an accumulator and a term of any type and returns a new accumulator and term upon success. All the one-layer combinators take a strategy of this type and return a strategy of the same type. They are defined with the help of the `gfoldl` function [8, 12] which works more or less the same way as list folding.

$$\texttt{gfoldl} \odot f \ (C\, t_1\, t_2 \ldots t_n) = f(C) \odot t_1 \odot t_2 \cdots \odot t_n$$

The unary operator f is applied to the constructor C, then the result is passed to the binary operator $\odot$ with the first subterm, obtaining a result which is again passed to the binary operator along with the second subterm, and so on. Thus `all_l` is defined as follows, and will be explained below.

```
newtype Xall_l a t = Xall_l {unXall_l :: Maybe (a,t)}

all_l :: GAT a -> GAT a
all_l s a t = unXall_l (gfoldl k z t)
  where z d = Xall_l (return (a, d))
        k (Xall_l x) t = Xall_l (do (a,d) <- x
                                    (a',t') <- s a t
                                    return (a', d t'))
```

If this looks awfully complicated, it is the auxiliary data type `Xall_l` that is to be blamed. Its sole purpose is to make the type system happy. Otherwise, the definition of `all_l` could be simplified as follows.

```
all_l s a0 = gfoldl k z
    where z d = return (a0,d)
          k x t = do (a,d) <- x
                     (a',t') <- s a t
                     return (a',d t')
```

Passed along the fold are an accumulator and a partially applied term, encapsulated in `Maybe`. A `Nothing` value indicates a failure in the previous computation and thus should be propagated (this is hidden by using the monad instance of `Maybe`). Otherwise, the value is passed to the binary operator `k` whose second parameter is the current subterm. `k` applies the visit to the current subterm resulting in a new accumulator and a new term. The partially applied constructor is applied to the changed term and is returned along with the new accumulator. The initial value for the fold is obtained from the unary operator `z` which, when applied to the data constructor, returns the initial accumulator and the constructor.

Having understood the logic, we can then examine the type of `gfoldl`, which is the reason why the above simplified code does not type-check.

```
gfoldl :: (forall a t. Data t => c (t -> a) -> t -> c a)
       -> (forall g. g -> c g)
       -> b -> c b
```

Understanding the above type signature is difficult. The first line is the type for the binary operator; the second line is the unary operator. It is not surprising to see that both operators have polymorphic types because they are applied to all direct subterms that do not necessarily have the same type. The term to fold is of type `b` and the result is of type `c b`. The same type constructor is used for the unary and binary operators. In the case of `all_l`, the pair whose type is `Maybe (a,b)` does not match the form `c b`. This is why the auxiliary data type is needed, that is, the type constructor `Xall_l a` plays the role of `c` here.

Defining a right-to-left traversal is more tricky, because no `gfoldr` is available. We need to do a left fold and incrementally generate a function along the fold.[2] The function, when applied to an accumulator, applies the traversal to the current term and the accumulator, and then passes the result to the function generated from the previous term.

```
newtype Xall_r a t = Xall_r {unXall_r :: a -> Maybe (a,t)}

all_r :: GAT a -> GAT a
all_r s a t = unXall_r (gfoldl k z t) a
    where z d = Xall_r (\a -> return (a,d))
          k (Xall_r g) t =
              Xall_r (\a -> do (a',t') <- s a t
                               (a'',d) <- g a'
                               return (a'',d t'))
```

The other two one-layer strategies `one_l` and `one_r` are slightly more involved, but can be defined similarly.

Now, to define the recursive traversal fulltd, we still need a sequential composition combinator, which can be defined as follows.

```
compose :: GAT a -> GAT a -> GAT a
compose s1 s2 a t = do (a',t') <- s1 a t
                       s2 a' t'
```

With `all_l` and `compose`, we are ready to define the top-down full-traversal strategy.

3.5 The Mother of All Traversals

Before we present the definition of the top-down full-traversal, let us first examine all the coverages we mentioned, namely, full, spine, stop, and once. If the horizontal direction is ignored, all the four variations can be summarized as follows.[3]

$$\begin{aligned}
\mathsf{fulltd}(v) &= v; \mathsf{all}(\mathsf{fulltd}(v)) \\
\mathsf{spinetd}(v) &= v; \mathsf{one}(\mathsf{spinetd}(v)) \\
\mathsf{stoptd}(v) &= v + \mathsf{all}(\mathsf{stoptd}(v)) \\
\mathsf{oncetd}(v) &= v + \mathsf{one}(\mathsf{oncetd}(v))
\end{aligned}$$

The choice combinator $+$ takes two strategies, and tries the first one. Only if it fails, the second one is applied. Since the visits return `Maybe` values, the choice combinator can be defined in Haskell as follows.

```
choice :: GAT a -> GAT a -> GAT a
choice s1 s2 a t = s1 a t `mplus` s2 a t
```

Now we can observe a strong similarity among all these traversal strategies: they all have the same form, the only differences being all/one and the ;/+ combinators. Examining the bottom-up versions reveals the same similarity:

$$\begin{aligned}
\mathsf{fullbu}(v) &= \mathsf{all}(\mathsf{fullbu}(v)); v \\
\mathsf{spinebu}(v) &= \mathsf{one}(\mathsf{spinebu}(v)); v \\
\mathsf{stopbu}(v) &= \mathsf{all}(\mathsf{stopbu}(v)) + v \\
\mathsf{oncebu}(v) &= \mathsf{one}(\mathsf{oncebu}(v)) + v
\end{aligned}$$

In fact, we can observe that these bottom-up strategies are just the flip side of the top-down strategies. Take this literally, replacing ; and $+$ with their flipped versions in the definitions of the top-down strategies, we obtain exactly the bottom-up counterparts. Thus, we can generalize the pattern and define a "mother of all traversals" [11] that can generate all these traversal strategies given appropriate parameters.

$$\mathsf{mother}(s) = s \cdot f(\mathsf{mother}(s))$$

The combinator f is a one-layer strategy, which can be either `one_l`, `one_r`, `all_l`, or `all_r`. The combinator $\cdot$ is taken from `compose`, `choice`, `compose'` and `choice'` where `compose'`

[2] This approach is called second-order fold [19, 26].

[3] stoptd is also called alltd in [22]

and `choice'` are the flipped versions, with the two parameters swapped.[4]

```
compose' s1 s2 = compose s2 s1
choice' s1 s2 = choice s2 s1
```

Each combination of parameters uniquely determines the behavior of the traversal. Table 4 lists all possible combinations.

	↓		↑		
	compose	choice	compose'	choice'	
all_l	full	stop	full	stop	→
one_l	spine	once	spine	once	→
all_r	full	stop	full	stop	←
one_r	spine	once	spine	once	←

Table 4. Children of the Mother of Traversals

With the mother of all traversals, traversals of different coverage, vertical, and horizontal directions are just a matter of partial applications of fixed parameters. The actual definition of `mother` in Haskell takes into consideration the context.

```
mother :: (GAT a -> GAT a -> GAT a) ->
          (GAT a -> GAT a) ->
          GCU c ->
          GCAT c a ->
          GCAT c a
mother g f u s c a t = (s c
                        `g`
                        f (mother g f u s (u c t))
                       ) a t
```

The context `c` is updated by the context updater `u` and passed to one-layer strategy combinator `f`.

The `mother` function is used to define instances of `traverse` by fixing the parameters `g` and `f` as shown in the next subsection.

3.6 Failure and Continuation

One issue worth mentioning is that a visit either fails or succeeds on a node. Continuation depends on the recursive traversal strategy. In the case of generic traversals, since the generic visits are converted from specific visits, there is in fact a third case. That is, none of the visits is applicable to the node. Handling such cases requires discretion from the designers. In our library, it is handled differently depending on the coverage of the traversal. In a full or spine-traversal, such cases are regarded as successful visits that do not change the term nor the accumulator. The rationale behind this is that users write specific visits and apply them everywhere applicable. If they want to stop a traversal, they should explicitly signal a failure. Under this assumption, the users are able to perform traversals even if they do not have complete knowledge of the whole tree. Therefore, in a full or spine-traversal, the traversal never fails unless a visit fails.

However, in a stop or once-traversal, a non-applicable visit is regarded as a failure. This is because in these two kinds of traversals, the traversal continues after failed visits. In a once or stop-traversal, the traversal succeeds only when there is a successful visit. Similarly, if a user does not have complete knowledge of the whole term, she is still able to handle those she is interested in and ignore others.

As we have seen in Section 3.5, we need to pass generic visit functions to the core combinator. However, the `traverse` function takes a list of specific functions. The gap is filled by type extension. Similar to the `mkT` and `mkQ` functions from [12], a generic function is used as the unit value for a fold operation over the list. The binary operator for the fold is the type extension function `ext0` defined in the `Data.Generics.Aliases` module of the Haskell Hierarchical Libraries [8]. The unit value is chosen based on the policy we just described. For full or spine-traversals, it is a function that always succeeds.

```
vsucc :: GAT a
vsucc a t = return (a,t)
```

For stop- or once-traversals, it is a function that always fails.

```
vfail :: GAT a
vfail _ _ = mzero
```

One of the two above combinators is chosen based on the coverage and used as a unit for the fold on the list of specific visits. In cases when the context updaters are present, they are also folded, with the unit being the constant function. The parameters `g` and `f` of the `mother` function presented above are chosen based on the coverage and traversal directions by looking up Table 4. For instance, the instance of the `traverse` function for contextual accumulating transformations is given as follows.

```
instance Data t => Traversal AccTrans Ctx c a t
                     ([GenU c a] -> [GenCAT c a]
                      -> c -> a -> t -> Maybe (a,t))
   where traverse _ _ cov vd hd us vs =
                  travt cov vd hd (foldC us)
                                  (foldV (catchv cov) vs)
```

The function `travt` looks up the table and partially applies `mother` to appropriate parameters.

```
travt :: Coverage -> VD -> HD -> GCU c
         -> GCAT c a -> GCAT c a
travt cov v h = mother (g cov v) (f cov h)
    where g :: Coverage -> VD -> GAT a -> GAT a -> GAT a
          g Full  FromTop    = compose
          g Spine FromTop    = compose
          g Once  FromTop    = choice
          g Stop  FromTop    = choice
          g Full  FromBottom = compose'
          g Spine FromBottom = compose'
          g Once  FromBottom = choice'
          g Stop  FromBottom = choice'
          f :: Coverage -> HD -> GAT a -> GAT a
          f Full  FromLeft  = all_l
          f Stop  FromLeft  = all_l
          f Spine FromLeft  = one_l
          f Once  FromLeft  = one_l
          f Full  FromRight = all_r
          f Stop  FromRight = all_r
          f Spine FromRight = one_r
          f Once  FromRight = one_r
```

`foldC` folds the specific context updaters. It begins with the unit (the `const` function), extends with the specific functions in the list. `foldV` does the same for the visits. However, for the visits, the unit will be determined by the coverage as we have just explained. This is realized by the function `catchv`, which determines the unit value for `foldV` as follows.

```
catchv :: Coverage -> GAT a
catchv Full  = vsucc
catchv Spine = vsucc
catchv Once  = vfail
catchv Stop  = vfail
```

Other instances of `traverse` are defined similarly. In cases where the context is not present, a default value for the context is needed. We use `undefined` since we need a value of type `a` and since it will never be accessed in a lazy evaluation setting. Transformations and

[4] We would have defined them using the `flip` function, but the type system prevented us from doing so, due to the rank-2 polymorphism.

accumulations are converted to accumulating transformations by providing a default implementation for the missing part and passed to the `mother` function and the result is converted back. We omit the tedious details here for simplicity.

3.7 Fixpoint Traversals

So far all the traversal strategies are one-pass strategies, which means that they apply a visit at most once to one node. Consider the case of beta reduction of lambda terms with applicative order. One step of reduction on a redex might result in a new redex inside the original one. A bottom-up traversal does not always result in a beta normal form. In such cases, an innermost traversal is needed. Such traversal strategies that apply visits to a term repeatedly until they are not applicable anymore are called fixpoint traversals. An innermost traversal applies a visit to an innermost subterm and obtains a new term. It repeats this process until no such subterm exists that the visit can be successfully applied. The innermost strategy is defined as follows [25].

$$\mathsf{innermost}(s) \quad = \quad \mathsf{repeat}(\mathsf{oncebu}(s))$$

Here the repeat combinator applies a strategy to a term until it fails.

Our library enables the definition in a typed framework. This combinator, along with several other primitive combinators are part of the library targeted for advanced users. So far, we have defined these combinators: a succ is a strategy that always succeeds without changing the term or the accumulator. This is the `vsucc` function we just defined. Note that it is also merely a curried version of the `return` function of the `Maybe` monad. Not very surprisingly, the strategy fail that always fails is the `vfail` function we defined in Section 3.6. The try strategy [25] takes another strategy and tries to apply it. If it fails, the succ strategy is used:

```
try :: GAT a -> GAT a
try s = s `choice` vsucc
```

Now, the repeat combinator [25] is defined in terms of try recursively.[5]

```
rep :: GAT a -> GAT a
rep s = try (s `compose` rep s)
```

Note that passing an identity transformation (one that always succeeds and returns the original term as the modified term) to repeat will cause an infinite loop. Notice that an outermost strategy is symmetric to innermost [25]:

$$\mathsf{outermost}(s) \quad = \quad \mathsf{repeat}(\mathsf{oncetd}(s))$$

Therefore, they both can be defined as instances of a more general xmost combinator with the help of mother.

```
xm :: (GAT a -> GAT a -> GAT a) ->
      (GAT a -> GAT a) ->
      GCU c ->
      GCAT c a ->
      GCAT c a
xm g f u s c = rep (mother g f u s c)
```

By choosing `g` from `choice` and `choice'` and `f` from `one_l` and `one_r`, innermost and outermost traversal strategies in both directions can be defined.

The aforementioned beta reduction application can be defined with innermost or outermost traversals depending on the reduction strategy. The following two Haskell functions implement applicative and normal-order beta reductions, respectively.

[5] To avoid name clash with `Prelude.repeat`, it is named `rep`.

```
appEval :: Lam -> Lam
appEval = innermost Trans NoCtx FromLeft
                    (reduce `whenever` isRedex)

normEval :: Lam -> Lam
normEval = outermost Trans NoCtx FromLeft
                     (reduce `whenever` isRedex)

isRedex :: Lam -> Bool
isRedex (App (Abs _ _) _) = True
isRedex _ = False
```

A visit reduces the term if it is an redex and fails otherwise. The innermost or outermost traversal strategy applies such a visit repeatedly to some subterm until it contains no redex anymore. A one-step reduction is performed by a full traversal searching for occurrences of the bound variable. A list of locally bound variables is passed as a context so that they are not substituted. The `reduce` function will be presented in Section 4.

4. Examples

In this section, we explore a few more sophisticated traversals and demonstrate how to implement them with our library. Suppose we again want to increase salaries in a company, but we only have a limited budget. We keep traversing the company data structure, increasing everyone's salary until the budget is all spent. The `incS` function then needs to know the total amount increased for the already visited people. This problem can be implemented by using an accumulating transformation. The remaining budget is passed along the traversal. Whenever we increase a salary, the increment has to be taken from the budget. The salary should not change if the budget is exhausted. The visit works on `Salary` values as did `incS`. The difference is that it returns a new budget paired with the changed salary.

```
incBud :: Data t =>
          Float -> Float -> t -> Maybe (Float,t)
incBud bud k = acctrans (always (incSbud k)) bud

incSbud :: Float -> Float -> Salary -> (Float,Salary)
incSbud k c (S s) = (c-i,S (s+i))
    where i = min (s*k) c
```

In this application, if the budget is exhausted, those who are visited later in the traversal (in this case, those at the right and the bottom) are left without an increase, which is not a fair strategy. A more sophisticated approach is to examine the salaries of all employees and the budget and then decide what to do with each individual salary. We can imagine different strategies. A socialisticly inclined increase would start increasing the lowest salaries first. In a capitalistic approach, we would start with the highest salaries. Any such scheme can be passed as a parameter to a smart increase function. The scheme is a function that takes a list of all salaries and returns a list of new salaries. The company data structure is traversed and the salaries are collected in a list passed to the scheme. The salaries are replaced with the ones in the new list. It appears that two passes are needed to accomplish the whole task. However, thanks to lazy evaluation, we can implement it with just one pass using a trick devised by Bird in 1984 [1, 5]. The visit, which is an accumulating transformer, works on `Salary` values. The old salary is appended to the salary list. A new salary is taken out of the new list and replaces the old salary. The new list is obtained by applying the scheme to the old salary list, which is just the first component of the result of the smart increase function. Since the visit never fails and the traversal is a full-traversal, we can safely assume that the return value is never `Nothing`.

```
incSmt :: Data t =>
          ([Float] -> [Float]) -> t -> ([Float],t)
incSmt scheme t = fromJust (acctrans (always v) [] t)
    where v a (S s) = (a++[s],S (new!!length a))
          new = scheme (fst (incSmt scheme t))
```

The above smart increase function provides endless possibilities. As an example, we show the capitalistic scheme as follows.

```
capitalism :: Float -> Float -> [Float] -> [Float]
capitalism bud k ys = ys3
     where (ys1,xs)  = ixSort ys [1..]
           (_  ,ys2) = foldr f (bud,[]) ys1
           (_  ,ys3) = ixSort xs ys2
           f s (b,ys) = let i = min (s*k) b
                        in (b-i,(s+i):ys)

ixSort :: Ord a => [a] -> [b] -> ([a],[b])
ixSort xs ys  =
  unzip $ sortBy (\(x,_) (y,_)->compare x y) $ zip xs ys
```

The list of all salaries is zipped with an index list `[1..]` and is sorted by the salaries. We then perform a right fold, which increases salaries sequentially from the right, to obtain a new salary list zipped with the indices. The result is then sorted again by the indices to recover the original order and unzipped. The socialistic scheme can be similarly defined using a left fold instead.

Now, let us consider the problem of beta reduction we brought up in Section 1.1. Our task is to implement a one-step beta reduction on a redex. This problem can be solved with a contextual transformation.

```
reduce :: Lam -> Lam
reduce (App (Abs v e) d) = fromJust (
       transformCB (mk upd) (always $ subst v d) [] e)
reduce e = e

upd :: [Name] -> Lam -> [Name]
upd bv (Abs v _) = v:bv
upd bv _ = bv

subst :: Name -> Lam -> [Name] -> Lam -> Lam
subst v d bv e@(Var (V v')) | v'==v && notElem v bv = d
subst _ _ _ e = e
```

The `reduce` function performs a bottom-up recursive transformation on the body of a beta redex. This context-sensitive transformation substitutes all the free occurrences of the formal parameter with the actual parameter. The context is a list of bound variables. It is updated by the `upd` function. The `subst` function takes the formal parameter, the actual parameter, a list of bound variable, and a term. If the term matches the formal parameter and is not bound, it is substituted by the actual parameter and otherwise unchanged.

5. A Practical Application

The library we have described in this paper has been successfully applied in a program transformation project that deals with Haskell programs [7]. The full Haskell abstract syntax consists of about 10 data types and at least 30-40 constructors in total. Repeatedly implementing recursions over such structures is tedious and non-modular. In the project, we needed several such recursions. The generic traversals greatly reduced the amount of code. Here is a simplified example of a recursion. In this function, we need to traverse expressions and replace the first subexpression that meets a certain criterion. The criterion relies on the variables bound by the surrounding environment. We not only need the changed expression but also the subexpression that was replaced. The type of this function is:

```
f :: [HsName] -> HsExp -> HsExp -> Maybe (HsExp,HsExp)
```

The arguments are: bound variables, new subexpression, and the expression to be transformed. The result is an optional pair of the changed expression and the original subexpression.

We can model the function as a once-traversal with a context being the bound variables and an accumulating transformation that does the replacing. The original subexpression replaced is returned as the result accumulator. We present the pseudo code to illustrate the essential use of the traversal function.

```
f bv ne e = onceAcctransC
                (mk cfe ++ mk cfd ++ mk cfm)
                (malways (qte ne))
                (bv,ls)
                undefined
                e
    where qte ne bv e =
              if  some condition bv e
               then Just (e,ne) else Nothing

          cfe bv (HsLambda ps _) =
              bv ++  variables bound in ps
          cfe bv (HsLet ds e) =
              bv ++  variables bound in ds
          cfe bv _ = bv

          cfd (bv,ls) (HsPatBind _ p _ ds) =
              bv ++  variables bound in p  and ds
          cfd c _ = c

          cfm bv (HsMatch ...) = ...
```

In this example, the context, which is given by the collection of bound variables, is changed whenever a binding is introduced. In Haskell, an expression, a declaration or a match can each introduce bindings. They are defined as different data types. Therefore, the generic context function needs to be composed of three specific cases. Functions `cfe`, `cfd`, and `cfm` are such specific functions.

6. Related Work

Without generic programming, functional programs suffer from a scalability problem (not necessarily efficiency wise, but with regard to the design). Generic functions whose behavior is defined inductively on the structures of the data can be scaled to large data structures easily without extra effort. They can even be reused for data types that are not yet defined. Our problem domain is program transformation and program generation, in particular, automatic monad introduction [7] and parameterized program generation [6]. Practical problems on large data structures such as the abstract syntax of Haskell and Fortran call for generic term traversals. Various approaches can be used for the purpose of generic term traversals. The program transformation tool Stratego/XT implements a set of strategies many of which are related to generic traversal [22]. However, the language lacks a strong static type system. Generic Haskell [4, 17] is a language extension to Haskell. It allows one to define purely generic functions. But a generic function is not a first-class citizen in Generic Haskell, which means that we can not define higher-order generic functions.

In [16] and [15], a combinator library (Strafunski) including generic traversal combinators is presented. These papers categorize a strategy into type preserving and type unifying strategies. To some extent, they correspond to the concepts of transformations and accumulators proposed in the present paper. A set of traversal schemes is also defined. These schemes, along with those defined in Stratego [23, 25, 2] are the main inspiration of our categorization of the problem. In [11] Lämmel proposed a highly parameterized generic traversal combinator. We implement these traversal

strategies in a statically typed framework proposed in [12, 13, 14]. Hinze, Löh, and Oliveria propose a *spine view* of data types and use it to define underlying SYB generic functions [9]. Because they are mostly compatible with the original SYB functions other than the embedded type information, this approach can be used to replace the underlying mechanism of creating generic transformations/accumulations as well.

Contextual visits are closely related to scoped dynamic rewrite rules [21, 3]. Dynamic rules are generated at run-time and can access their context. A scope can be imposed to remove rules after they are not valid anymore. One problem with scoped dynamic rules is that it is necessary to inline the definition of the traversal strategy so that the scope can be included in the traversal of subterms. The approach therefore suffers from a modularity problem. In our library, context is abstracted and modularized. It is taken care of by the recursive traversal strategy and passed to the visit so that the visit does not need to worry about the scope.

In [20], van den Brand et al. categorize a traversal into transformation, accumulation, and accumulating transformation. This agrees with our categorization. In fact, we borrowed these terms from [20]. They also identify certain properties of traversals and place them in the corresponding positions in the "traversal cube". We have enriched the cube by extending the coverage axis.

7. Summary and Future Work

In this paper, we extended the scrap-your-boilerplate approach proposed by Lämmel and Peyton Jones. We have analyzed the problem domain of generic traversals and have extracted five orthogonal parameters of a traversal. We have defined one universal generic traversal combinator that can be parameterized to cover the whole problem domain space. In summary, these combinators provide the programmers these choices:

- The *visit*. We can perform a transformation that modifies a node, an accumulation that gathers information from nodes along the traversal, or an accumulating transformation that does both.
- The *context*. The action might rely on the path from the root node to the current node. A customized context can be maintained by a context updater function and carried to the visit function.
- The *vertical traversal order*. A traversal can start from the top of the term and moves down or the opposite direction.
- The *horizontal traversal order*. A traversal can visit from left to right or the opposite direction.
- The *coverage*. A traversal can visit all the nodes, bypass children of certain nodes, visit along a spine from the root to a leaf, or stop after a successful visit.

The clients can easily choose the appropriate strategy and focus on the "interesting parts", the recursion is performed by the generic traversal combinators.

In addition to this high-level interface, we have also defined a set of primitive combinators that can be used to define additional recursive traversal strategies.

However, our library only addresses the problem of transformations and accumulations on one term. Problems that involve parallel traversing two terms such as generic zip [13, 10] cannot be handled. Although these combinators are fairly general, there is still room for improvement. Regardless of the two traversal directions, we always favor the vertical direction over the horizontal direction, which means we always implement a depth-first traversal. One possible extension is to have symmetric breadth-first traversals. Moreover, we only have `one` and `all` strategies as our one-layer strategies. We can also consider strategies that visit only some of of direct subterms of a term. We believe these features will extend the traversal space and complement the traversal library.

Acknowledgments

We would like to thank Ralf Lämmel for discussing with us initial ideas of the presented library during a visit at Oregon State University. We also thank the anonymous reviewers for their helpful feedbacks.

References

[1] R. S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, 21:239–250, 1984.

[2] M. Bravenboer, K. Trygve Kalleberg, R. Vermaas, and E. Visser. *Stratego/XT Tutorial, Examples, and Reference Manual (latest)*, 2006. `http://nix.cs.uu.nl/dist/stratego/strategoxt-manual-unstable-latest/manual`.

[3] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundamenta Informaticae*, 69:1–56, 2005.

[4] D. Clarke and A. Löh. Generic Haskell, Specifically. In J. Gibbons and J. Jeuring, editors, *IFIP TC2 Working Conference on Generic Programming*, 2002.

[5] O. de Moor. An Exercise in Polytypic Program Derivation: *repmin*. Unpublished `http://web.comlab.ox.ac.uk/oucl/work/oege.de.moor/papers/repmin.ps.gz`, 1996.

[6] M. Erwig and Z. Fu. Parametric Fortran – A Program Generator for Customized Generic Fortran Extensions. In *6th Int. Symp. on Practical Aspects of Declarative Languages*, LNCS 3057, pages 209–223, 2004.

[7] M. Erwig and D. Ren. Monadification of Functional Programs. *Science of Computer Programming*, 52(1–3):101–129, 2004.

[8] *Haskell Hierarchical Libraries*. `http://www.haskell.org/ghc/docs/latest/html/libraries/`.

[9] R. Hinze, A. Löh, and B. Oliveira. "Scrap Your Boilerplate" Reloaded. In P. Waldler and M. Hagiya, editors, *8th International Symposium on Functional and Logic Programming*, pages 24–26, 2006.

[10] G. Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[11] R. Lämmel. The Sketch of a Polymorphic Symphony. In Gramlich B. and Lucas S., editor, *Proc. of International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002)*, ENTCS 70. Elsevier Science, 2002.

[12] R. Lämmel and S. Peyton Jones. Scrap Your Boilerplate: a Practical Design Pattern for Generic Programming. In *Types in Language Design and Implementation*, volume 38, pages 26–37, 2003.

[13] R. Lämmel and S. Peyton Jones. Scrap More Boilerplate: Reflection, Zips, and Generalised casts. In *9th ACM International Conference on Functional Programming*, pages 244–255, Snow Bird, UT, USA, 2004.

[14] R. Lämmel and S. Peyton Jones. Scrap Your Boilerplate with Class: Extensible Generic Functions. In *10th ACM International Conference on Functional Programming*, pages 204–215, Tallinn, Estonia, 2005.

[15] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. Technical Report SEN-R0124, Centrum voor Wiskunde en Informatica, 2001.

[16] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *4th Symposium on Practical Aspects of Declarative Languages*, LNCS 2257, pages 137–154. Springer-Verlag, 2002.

[17] A. Löh, J. Jeuring, et al. The Generic Haskell User's Guide. Technical report, Utrecht University, 2005.

[18] Reclib. A Recursion and Traversal Library for Haskell. `http://eecs.oregonstate.edu/~erwig/reclib/`.

[19] T. Sheard and L. Fegaras. A Fold for All Seasons. In *6th Conference on Functional Programming and Computer Architecture*, pages 233–242. ACM Press, 1993.

[20] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term Rewriting with Traversal Functions. *ACM Transactions on Software Engineering and Methodology*, 12(2):152–190, 2003.

[21] E. Visser. Scoped Dynamic Rewrite Rules. In van den Brand M. and R. Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *ENTCS*. Elsevier Science Publishers, 2001.

[22] E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In Lengauer C. et al., editors, *Domain-Specific Program Generation*, LNCS 3016, pages 216–238. Springer-Verlag, 2004.

[23] E. Visser. A Survey of Strategies in Rule-Based Program Transformation Systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.

[24] E. Visser and Z.-e.-A. Benaissa. A Core Language for Rewriting. In C. Kirchner and H. Kirchner, editors, *2nd International Workshop on Rewriting Logic and its Applications*, ENTCS 15, 1998.

[25] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *3rd ACM International Conference on Functional Programming*, pages 13–26, 1998.

[26] M. Wand. Continuation-Based Program Transformation Strategies. *Journal of the ACM*, 27(1):164–180, 1980.