# Generic Representations of Tree Transformations

Jeroen Bransen and José Pedro Magalhães

16 July 2015

**Abstract**

Applications that deal with editing of structured data over multiple iterations, such as structure editors, exercise assistants, or applications which support incremental computation, need to represent transformations between different versions of data. A general notion of "transformation" should be more informative than what is obtained by computing the difference between the old and the new term, as diff algorithms generally consider only insert, copy, and delete operations. Transformations, however, may involve swapping elements, or duplicating them, and a good representation of transformations should not involve unnecessary repetition of data, and should maximize sharing between the original and the transformed term.

In this paper we take a detailed look at the notion of *transformation* on strongly-typed structures. Our transformations are datatype-generic, and thus can be applied to a large class of data structures. We focus on representing transformations in a way that maximally captures the common substructure between the old and new term. This is of particular importance for incremental computations which recompute information only for the parts of the tree that are affected by a transformation.

We present a library for encoding such transformations over families of data types, together with an algorithm for computing a transformation from one term into another, while retaining shared common substructures. We provide practical examples of how to encode transformations, as well as a realistic application to computing transformations between different revisions of a program.

## 1 Introduction

Structured data abounds: from linked lists and binary trees, to abstract syntax trees (ASTs) and expression languages, most computer data has some form of structure. Programmers often choose not to encode this structure rigidly, and live with the consequences of trading flexibility for safety. However, in a statically-typed language such as Haskell (Peyton Jones, 2003), the structure of values can be enforced by the type checker, and programmers generally take advantage of this to help guarantee the correctness of their code.

Large applications, besides manipulating structured data, often need to *transform*, *edit*, or *evolve* this data. Let us consider some concrete examples:

**Structure editors** A structure editor is a type of editor that is aware of the underlying structure of the document being edited. Text editors are *not* structure editors; Proxima (Schrage, 2004) is a good example of a structure editor. In such editors, the underlying structure of the data being edited is made visible to the user. As such, care has to be taken to ensure that edit operations are kept efficient; while copy-pasting thousands of lines of text might be fast enough even if the data is simply duplicated, duplicating thousands of nodes in an AST might lead to unacceptable delays for the user of the structure editor. It is preferable to encode edits as transformations from a previous document into a newer one, keeping track only of what exactly has changed, while reusing the rest.

**Exercise assistants** An exercise assistant is a tool to help students understand and apply fundamental concepts in a given domain. An example of an exercise assistant is Ask-Elle (Gerdes, 2012), a Haskell tutor designed to help students develop functional programs incrementally, while giving hints and semantically rich feedback on their progress. Here, too, the notion of transformation is important, because writing a simple program, or solving a linear equation, is generally a step-wise process, consisting of basic laws applied at specific locations. It is more efficient to track each successive transformation done by the user than to recompute a difference between an old and newer term each time. Also, a traditional difference analysis might fail to adequately track operations such as swapping items in a list, and this, in turn, can hurt the quality of the feedback offered to the student.

**Incremental computations** To avoid expensive recomputation in unchanged parts of data, an incremental computation keeps track of changes (Reps *et al.*, 1983; Acar, 2005). An example is the incremental evaluation of attribute grammars using change propagation (Bransen *et al.*, 2015): changing a value somewhere inside a tree might not require recomputing all the attributes. As such, incremental computations need information about how terms are transformed, and the quality of this information affects the recomputation avoidance mechanism.

In this paper we tackle the problem of representing transformations on values in such a way that applications as those described above can exploit this representation to improve their functionality. We look at the problem from a datatype-generic perspective (Gibbons, 2007), such that our description of transformations is strongly-typed and applicable to a large class of data types. We are not exclusively interested in computing a *difference* between two terms, as Lempsink *et al.* (2009) did. Instead, we focus on the more general notion of encoding *transformations* as they happen (as captured, for example, by a graphical user interface), and representing these transformations in a way that minimises the duplication of data. We implement this in the functional programming language Haskell.

Specifically, our contribution is in identifying the need for more precise representations of transformations, and giving a concrete solution. All the code described in this paper, together with examples and benchmarks, is available at `http://dreixel.net/research/code/transformations.tar.gz`. The library itself is available at `http://hackage.haskell.org/package/transformations`.

This paper is based on an earlier conference publication (Bransen & Magalhães, 2013). The main difference is that we focus on a single representation of transformations, from which we have removed runtime type comparisons (and thus improved performance). We improved the approach with a well-typed representation of paths based on zippers, pattern synonyms for convenience of usage, and provided a larger example application.

The rest of this paper is organised as follows. We begin by identifying transformation operations and motivating the problem we tackle in Section 2. Our solution is generic, but we start explaining our approach with a non-generic solution in Section 3. Section 4 provides a brief introduction to generic programming with regular functors, and Section 5 describes how to encode paths in a data type by using generic one-hole contexts. We proceed by detailing our generic representation of transformations in Section 6, followed by its generalisation to handle mutually recursive families of data types in Section 7. Section 8 provides an application of our approach to tracking transformations between versions of a LUA program. We conclude in Section 9, also discussing possible improvements to our approach and some related work.

## 2 Transformation operations

In this section we show a number of transformations that we are interested in encoding. We do this by showing example transformations on expressions encoded by a simple data type:

```
data Expr = Var    String
          | Const  Int
          | Neg    Expr
          | Add    Expr Expr
```

An expression is either a named variable, an integer constant, the negation of an expression, or the addition of two expressions. The following are sample expressions:

```
expr1, expr2, expr3 :: Expr
expr1 = Add (Const 1) (Var "a")
expr2 = Add (Const 1) (Neg (Var "a"))
expr3 = Add (Var "a") (Const 1)
```

We will use these sample expressions in examples given throughout this paper.

We now consider some typical transformations on expressions.

**Insertion** An insertion is a simple transformation that extends a value. Consider the following transformation:

```
Var "a" ⇝ Neg (Var "a")
```

The arrow "⇝" is used to indicate a transformation, with the old term on the left and the new term on the right. The right-hand side of this transformation can be seen as

arising from the insertion of *Var* `"a"` into the expression *Neg* _, where the underscore is here used informally to denote a hole in an expression. Our example $expr_1$ can be transformed into $expr_2$ using such an insertion, which happens in the context of the second subtree of the full expression.

**Replacement** In our setting, replacement is just insertion at a given location. Due to the way we represent transformations the newly inserted value always replaces the existing value, thereby making replacement a trivial transformation.

**Deletion** A deletion removes part of an expression. For example, $expr_1$ can be seen as arising from the deletion of the *Neg* constructor in $expr_2$. In reality, however, we see deletion as a form of replacement: $expr_1$ arises by replacing the *Neg* $x$ expression by $x$ in $expr_2$. We do not consider deletion to be a valid transformation because, in general, it results in ill-typed expressions.

**Swap** Insertion and deletion are edit operations considered by general tree difference algorithms, such as that of Lempsink *et al.* (2009). Consider now the transformation from $expr_1$ to $expr_3$, which has the following shape:

$$Add\ a\ b \rightsquigarrow Add\ b\ a$$

It is possible to encode this transformation with insertion and deletion operations alone. However, that approach has two drawbacks. For starters, it is verbose, requiring both a deletion and an insertion. Moreover, it does not adequately encode the fact that the subexpressions $a$ and $b$ remain unchanged through the transformation and do re-appear in the result. This problem is particularly relevant when the subexpressions being swapped are large and other computations depend on it, for example in a structure editor where the layout of the subtrees has already been computed and does not change due to the swap.

**Rotation** A rotation transformation involves rearranging the nesting structure of a tree. A common example is rebalancing binary operators, exploiting associativity:

$$Add\ a\ (Add\ b\ c) \rightsquigarrow Add\ (Add\ a\ b)\ c$$

In rotations, we want to keep track of the fact that some subexpressions (in our example, $a$, $b$, and $c$) are unchanged, and simply rearranged in their ancestors. Although swap also falls under this definition of rotations, we mention it separately because it is used as an example throughout the paper.

**Duplication** Duplication is a transformation typically arising from a copy-paste operation in an editor. For example:

$$Add\ a\ (Const\ 0) \rightsquigarrow Add\ a\ a$$

In this transformation, the subexpression $a$ has been duplicated. This is not the same as just inserting $a$, as we want to remember that the inserted subexpression is not new, but

just a copy of something already existing. The encoding of duplication is particularly interesting in the context of incremental evaluation, where values computed over $a$ can be preserved after a copy-paste operation due to the information of the two subtrees being identical.

## 2.1 Localisation

A concept that is relevant to all types of transformation is that of *localisation*. Consider the following transformation:

$$\begin{aligned} &\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Const}\ 1)))))))\\ \rightsquigarrow\ &\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Const}\ 2)))))))\end{aligned}$$

A good encoding of this transformation should not repeat the entire spine of *Neg* constructors. Instead, it should be able to represent the changes in a local fashion, focusing on a part of the tree only.

## 2.2 Diff is not enough

At this stage, it is worth looking at existing solutions and debating over whether they already provide a good solution to the problem we are tackling. The "standard" way of tracking changes between values is to use a diff algorithm. Lempsink *et al.* (2009) describe a type-safe, datatype-generic diff that may be used to determine changes in terms: given a transformation $t_1 \rightsquigarrow t_2$, *diff* $t_1$ $t_2$ returns an *edit script* describing how to transform $t_1$ into $t_2$. An associated *patch* operation can be used to apply an edit script to a term, obtaining a transformed term.

However, standard edit scripts only contain copy, insert, and delete operations. While these suffice to describe every transformation, the resulting edit description is often not faithful to the actual change that occurred. This is easily seen in a swap transformation, which, in an edit script, is represented by deletion and insertion. As an example, take the edit script resulting from computing the difference between *Add* (*Var* `"a"`) (*Var* `"b"`) and *Add* (*Var* `"b"`) (*Var* `"a"`):

$$\begin{aligned} &\mathit{Cpy}\ \mathit{Add}\ \$\ \mathit{Cpy}\ \mathit{Var}\ \$\ \mathit{Ins}\ \texttt{"b"}\ \ \$\ \mathit{Ins}\ \mathit{Var}\ \$\\ &\mathit{Cpy}\ \texttt{"a"}\ \$\ \mathit{Del}\ \mathit{Var}\ \ \$\ \mathit{Del}\ \texttt{"b"}\ \$\ \mathit{End}\end{aligned}$$

This edit script does not keep track of the fact that the inserted expressions are not "new", losing adequate sharing between transformations. We could extend existing diff algorithms with a swapping operation, but this would not be enough to capture rotation, or duplication. Trying to add new edit operations to capture each different transformation we can think of is tiresome, and we would have no guarantee that we covered all possible transformations. As such, we instead try to take a more general approach to the concept of transformation, remaining as abstract as possible as to what type of transformations are allowed, but making sure that sharing of subexpressions is kept explicit, with minimal duplication of information.

## 2.3 Overview

Intuitively, transformations are encoded as a list of localised insertions in which parts of the original tree can be reused. We use *trees with references* for the inserted values, where the references point to parts of the original tree. The insertions are paired with a path describing the location in the tree where the insertion should happen; the full transformation is then a list of those pairs. The insertions are applied one by one in the order they appear in the list, but the references always point to the original tree.

To represent trees with references we add a *Ref* constructor of type $Path \to Expr$ to the data type. For now, we abuse notation to introduce the idea informally, deferring the actual implementation to later sections. As an example, the transformation from $expr_1$ to $expr_2$ is expressed as follows:

```
insert :: Maybe Expr
insert = apply addNeg expr1 where
  addNeg :: Transformation Expr
  addNeg = [(Add1 End, Neg (Ref (Add1 End)))]
```

The first element of the tuple is a path indicating where the transformation takes place, with the number giving the child index. In this case the transformation thus takes place in the right child of the root node. The second element of the tuple is the value to be inserted at this location, in this case a *Neg* constructor with a reference. *Ref*'s indicate reuse, and contain a path to an element in the original tree. This reused part is again the right child of the root node (which, before the transformation, is simply *Var* `"a"`).

The transformation from $expr_2$ to $expr_1$ by deletion is encoded as follows:

```
delete :: Maybe Expr
delete = apply delNeg expr2 where
  delNeg :: Transformation Expr
  delNeg = [(Add1 End, Ref (Add1 (Neg0 End)))]
```

Here we first focus on the second child of *Add*. At that location, we insert a reference that points to a smaller part of the original subtree. This encodes the notion that the *Neg* constructor that was "in between" is deleted. Here $Add_1\ End$ refers to the second child again, which is the *Neg* constructor, and $Add_1\ (Neg_0\ End)$ to the first child of the *Neg* constructor, which is *Var* `"a"`.

The swapping operation is represented by two separate insertions, one to replace the left subtree by the right one, and the other to replace the right subtree by the left one:

```
swap :: Maybe Expr
swap = apply swap' expr1 where
  swap' :: Transformation Expr
  swap' = [(Add0 End, Ref (Add1 End))
          ,(Add1 End, Ref (Add0 End))]
```

In this example it becomes clear that it is essential that the references point to parts of the original tree, instead of using the already modified tree. After performing the

first insertion, the left and right child of the root are equal, so the part that needs to be inserted in the right child does not exist anymore in this intermediate tree. This is the reason why the paths used for references are different from the paths used to describe the location of the insertion; the former always refer to the original tree, whereas the latter refer to the intermediate state of the tree after some insertions have already been performed.

# 3 Lua example

Before we explain the generic approach to this problem we illustrate the representation in a non-generic way with a small example. We show a small subset of the data types for representing programs in the Lua programming language. The family of data types encoding Lua programs consists of 12 data types, containing a total of 60 constructors. We chose Lua because it is a language used in practice, and it has a moderately-sized AST. Furthermore, we can use an existing Haskell parser for Lua[1].

In this section we show the representation for just three of those constructors, from two different data types. The constructors are defined as follows:

```
data Stat = Do    Block
          | While Exp   Block
data Exp  = Binop Binop Exp Exp
```

The *Stat* and *Exp* datatypes contain many more constructors that we elide for simplicity.

## 3.1 Paths

The first component of the representation is the location of a term in a tree. For this we encode a path from the root node with a data type named *Path*:

```
data Path α β where
  End     ::                    Path β    β
  Do0     :: Path Block β    → Path Stat β
  While0  :: Path Exp   β    → Path Stat β
  While1  :: Path Block β    → Path Stat β
  Binop0  :: Path Binop β    → Path Exp  β
  Binop1  :: Path Exp   β    → Path Exp  β
  Binop2  :: Path Exp   β    → Path Exp  β
```

The *End* constructor marks the end of the path. The other constructors indicate that we are at a certain node (*Do*, *While* or *Binop*) and take a step into one of its children, for which the rest of the path is given. In general, the *Path* data type contains an *End* constructor and a constructor for each recursive child of each constructor in the family; for non-recursive children such as *Int* there is no constructor in the *Path* data type, as paths only point to recursive positions.

[1] `http://hackage.haskell.org/package/language-lua`

The paths are annotated with type information for the source and target of the path, as we have to represent paths through the different types of nodes in the AST using a single *Path* data type. We therefore define *Path* as a GADT, such that *Path* $\alpha$ $\beta$ represents a path in a tree of type $\alpha$ pointing to a node of type $\beta$. In a complete path the $\alpha$ parameter equals the type of the root of the AST.

## 3.2 Trees with references

The second component of our representation is the concept of trees extended with references. These are encoded by extending each data type in the family with a *Ref* constructor. Each constructor's name is postfixed with *E* to denote that it is the "extended" version. In this non-generic example, each of the *Ref* constructors is prefixed with the name of the corresponding data type in order to avoid name clashes.

```
data Stat_E τ = StatRef (Path τ Stat)
              | Do_E    (Block_E τ)
              | While_E (Exp_E   τ) (Block_E τ)
data Exp_E τ = ExpRef (Path τ Exp)
             | Binop_E (Binop_E τ) (Exp_E   τ) (Exp_E τ)
```

The type parameter $\tau$ gives the type of the full AST. The path stored in a reference is a path from the top of the AST towards a node of the type of that reference.

## 3.3 Complete transformation

We can now represent an insertion as a pair of a path—giving the location of the insertion—and a tree with references—giving the replacement. However, as the type of the replacement depends on the type of the node that the path points to, we can not directly specify this as a pair. Instead, we use a type family that maps the type $\alpha$ (for example *Stat*) to the corresponding type of trees with references (for example $Stat_E$).

```
type family   ReplType α   :: ⋆ → ⋆
type instance ReplType Stat = Stat_E
type instance ReplType Exp  = Exp_E
```

Note that in the generic setting such type family is not needed; we only present it here for sake of completeness of the non-generic example.

Using this type we can represent an insertion, which is also parametrised over the type of the top level node. We use a GADT to ensure that the type of the element being inserted matches what is expected at that position:

```
data Insert τ where
  Insert :: Path τ α → ReplType α τ → Insert τ
```

Finally, a full transformation is a list of such changes, parametrised over the type of the top level node:

```
type Transformation τ = [Insert τ]
```

Note that different elements in this list can insert nodes of different types.

# 4 Generic programming for regular functors

To tackle the problem of representing transformations generically we first need to introduce a library for generic programming, which we use for developing our solution. As we will see in the next section, our approach revolves around annotating recursive positions in data types. As such, using a library with an explicit encoding of recursion (i.e. with a fixed-point view (Holdermans *et al.*, 2006) on data) suits us best. We can either pick `regular` (Van Noort *et al.*, 2008), a library which supports only regular data types, or `multirec` (Rodriguez Yakushev *et al.*, 2009), a generalisation of `regular` that supports mutually-recursive families of data types. For presentation purposes, we use `regular`, as it is easier to understand our solution in the single-datatype case. We have also written an implementation using `multirec`, which we describe in Section 7.

This section provides only a brief introduction to `regular`. For more details, the reader is referred to Van Noort *et al.* (2008).

## 4.1 Representation

Data types are encoded in `regular` using the following five *representation types*:

$$\begin{array}{ll}
\textbf{data}\ U & \rho = U \\
\textbf{data}\ I & \rho = I\ \rho \\
\textbf{data}\ K\ \alpha & \rho = K\ \alpha \\
\textbf{data}\ (\phi :+: \psi) & \rho = L\ (\phi\ \rho) \mid R\ (\psi\ \rho) \\
\textbf{data}\ (\phi :\times: \psi) & \rho = \phi\ \rho :\times: \psi\ \rho
\end{array}$$

Unit, encoded by $U$, is used for constructors without arguments. Recursive positions, encoded by $I$, denote occurrences of the data type being defined. Constants, encoded by $K$, are used for all other constructor arguments. Sums, encoded by $:+:$, are used to denote choice between constructors, while products, encoded by $:\times:$, are used for constructors with multiple arguments. The `regular` library also contains representation types for dealing with data type meta-information such as constructor and selector names, but we elide those from our presentation as they are not essential.

As an example, the *Expr* data type of Section 2 is encoded in `regular` as follows:

$$\textbf{type}\ Expr_{PF} = K\ String :+: K\ Int :+: I :+: (I :\times: I)$$

Note that $Expr_{PF}$ (of kind $\star \to \star$) encodes the *pattern functor* of *Expr*, also known as its *open* version. To obtain *Expr*, we need to "close" $Expr_{PF}$, replacing the recursive positions under $I$ with $Expr_{PF}$ again. This can be done using a type-level fixed-point operator:

$$\textbf{data}\ \mu\ \phi = In\ (\phi\ (\mu\ \phi))$$

Now, $\mu\ Expr_{PF}$ encodes a data type that is isomorphic to *Expr*.

## 4.2 Functoriality of the representation types

The `regular` library encodes data types as *functors*; the recursive positions are abstracted into a parameter $\rho$. As such, we can provide *Functor* instances for the repre-

sentation types. These are unsurprising, with the action being transported across sums and products, ignored in units and constants, and applied at the recursive positions:

```
instance Functor U where
  fmap _ U = U
instance Functor (K α) where
  fmap _ (K x) = K x
instance Functor I where
  fmap f (I r) = I (f r)
instance (Functor ϕ, Functor ψ) ⇒ Functor (ϕ :+: ψ) where
  fmap f (L x) = L (fmap f x)
  fmap f (R x) = R (fmap f x)
instance (Functor ϕ, Functor ψ) ⇒ Functor (ϕ :×: ψ) where
  fmap f (x :×: y) = fmap f x :×: fmap f y
```

This functoriality can be used to define catamorphisms over the representation types.

## 4.3 Embedding user-defined types

To provide a convenient interface for generic functions, `regular` uses a type class to aggregate generic representations of user data types. This class defines how to represent each data type, and how to convert to and from its representation:

```
class Regular α where
  type PF α :: ⋆ → ⋆
  from :: α → PF α α
  to   :: PF α α → α
```

The type family *PF* encodes the pattern functor of the data type being represented. The conversion functions *from* and *to* do not operate on "fully generic" representations of type $\mu\,(PF\ \alpha)$. Instead, they operate on representations that are generic on the top layer, containing values of the original data type at the recursive positions (of type $\alpha$). This decision is taken merely for efficiency reasons, since now generic functions are non-recursive, and thus easier to optimise by inlining (Magalhães, 2013).

We can now complete our encoding of *Expr* in `regular`:

```
instance Regular Expr where
  type PF Expr = Expr_PF
  from (Var s)     = L (K s)
  from (Const i)   = R (L (K i))
  from (Neg e)     = R (R (L (I e)))
  from (Add e1 e2) = R (R (R (I e1 :×: I e2)))
  to (L (K s))                 = Var s
  to (R (L (K i)))             = Const i
  to (R (R (L (I e))))         = Neg e
  to (R (R (R (I e1 :×: I e2)))) = Add e1 e2
```

Instances of the *Regular* class are tedious to write by hand; fortunately, the `regular` library includes Template Haskell code to automatically generate these instances for user data types.

### 4.4 Generic functions

We can now define generic functions by giving a case for each representation type. We use a type class for this purpose, followed by five instances. As an example, here is the generic function that lists all the immediate children of a given term:

```
class Children φ where
  gchildren :: φ ρ → [ρ]

instance Children U where
  gchildren _ = []
instance Children I where
  gchildren (I x) = [x]
instance Children (K α) where
  gchildren _ = []
instance (Children φ, Children ψ) ⇒ Children (φ :+: ψ) where
  gchildren (L x) = gchildren x
  gchildren (R x) = gchildren x
instance (Children φ, Children ψ) ⇒ Children (φ :×: ψ) where
  gchildren (x :×: y) = gchildren x ++ gchildren y
```

The function *gchildren* operates on generic representations. We also define the function *children* which operates directly on user data types, by first converting them to generic representations:

```
children :: (Regular α, Children (PF α)) ⇒ α → [α]
children = gchildren ∘ from
```

The call *children expr*$_1$, for example, returns $[Const\ 1, Var\ \texttt{"a"}]$, as expected. The *to* function is not used here because *from* leaves the recursive positions unchanged.

## 5 Zippers and paths

The zipper is a data structure used to represent traversals in a term. It is a type-indexed data type (Hinze *et al.*, 2002): every algebraic data type induces a zipper, generically. We provide a brief introduction to zippers in this section because they form a key part of our solution. A detailed description, however, is out of the scope of this paper; Rodriguez Yakushev *et al.* (2009) describe a zipper for families of data types.

For now we focus on a zipper for regular functors. The zipper encodes a position of focus on a value, together with the surrounding context. These two elements are stored in the *Loc* data type:

```
data Loc α where
  Loc :: (Regular α) ⇒ α → [Ctx (PF α) α] → Loc α
```

A location is the point currently in focus in the zipper (of type $\alpha$), and the path to the focal point. This path is stored as a stack of one-hole *contexts*. The context is given by the derivative of the pattern functor representing the data type (McBride, 2001). This type-indexed data type is encoded in `regular` as a data family, indexed over the five representation types:

```
data family Ctx (φ :: ⋆ → ⋆) :: ⋆ → ⋆
data instance Ctx U          ρ
data instance Ctx (K α)      ρ
data instance Ctx I          ρ = C_Id
data instance Ctx (φ :+: ψ) ρ = C_L (Ctx φ ρ)
                              | C_R (Ctx ψ ρ)
data instance Ctx (φ :×: ψ) ρ = C_1 (Ctx φ ρ) (ψ ρ)
                              | C_2 (φ ρ)     (Ctx ψ ρ)
```

Units and constants contain no recursive positions, and as such have an empty context. The $C_{Id}$ constructor signals a recursive position. Sums can either have a context on the left ($C_L$) or on the right ($C_R$). For products, we can choose to traverse the first argument, keeping the second argument intact ($C_1$), or to do the opposite ($C_2$).

## 5.1 Paths

Zipper operations, such as navigation functions, can be defined on the type of contexts. However, we are only interested in the type of contexts, and will not use any zipper operations, so we do not go into further details about those. Instead, we show how we use zipper contexts to encode paths in data structures. A context instantiated with units for the recursive positions ($\rho$ set to $()$) effectively encodes a direction of navigation on a data structure to one of its children (Gibbons, 2013). Paths on a data type are then a list of such directions on the corresponding pattern functor:

```
type Dir  φ = Ctx φ ()
type Path α = [Dir (PF α)]
```

As an example, recall the *Expr* type. The following are all the possible navigation directions on this data type (modulo $\bot$):

```
neg_0, add_0, add_1 :: Dir (PF Expr)
neg_0 = C_R (C_R (C_L C_Id))
add_0 = C_R (C_R (C_R (C_1 C_Id (I ()))))
add_1 = C_R (C_R (C_R (C_2 (I ()) C_Id)))
```

These encode navigating to the first child of a negation, the first child of an addition, and the second child of an addition, respectively. Paths, being lists of directions, do not need any special helper syntax to be defined. For example, $[add_1, neg_0]$ is the path that traverses to *Const* 2 in the expression *Add* (*Const* 0) (*Neg* (*Const* 2)).

### 5.1.1 Pattern synonyms for directions

Since directions are type-indexed data types defined on pattern functors, they are cumbersome to build, and require knowledge of the particular encoding of the data type at hand (i.e. its *Regular* instance). Values like $neg_0$ can be seen as smart constructors for directions, and partially mitigate the problem of exposing directions to end users. However, pattern matching on directions is not improved by these smart constructors.

Fortunately, we can make use of the recently introduced "pattern synonyms" GHC extension[2] for this. Pattern synonyms allow us to define shorthands (synonyms) for constructors, much like we can use type synonyms as shorthands for types. Since we will generally be working with paths rather than directions, we define pattern synonyms for paths that navigate through each of the possible directions on *Expr*:

```
pattern End :: Path Expr
pattern End = []

pattern Neg0 :: Path Expr → Path Expr
pattern Neg0 x = CR (CR (CL CId)) : x

pattern Add0 :: Path Expr → Path Expr
pattern Add0 x = CR (CR (CR (C1 CId (I ())))) : x

pattern Add1 :: Path Expr → Path Expr
pattern Add1 x = CR (CR (CR (C2 (I ()) CId))) : x
```

With these pattern synonyms, $Add_1\ (Neg_0\ End)$ encodes the same path as $[add_1, neg_0]$, but has the advantage that it can be matched against.

# 6 Transformations with shared subexpressions

Knowing how to represent data types and paths generically, we are ready to describe our encoding of transformations, which provides a suitable interface both for generating and inspecting transformations.

## 6.1 Representation

The first part of the representation of transformations is the notion of paths in a tree. We represent paths using a zipper context as explained in Section 5. This is an improvement over our previous solution (Bransen & Magalhães, 2013) where we encoded paths as a list of integers. Our new encoding guarantees that we can only define valid paths (although they might not necessarily be valid for a given value; see Section 9.2).

To represent trees with references we extend the pattern functor of a type $\alpha$ to allow for references at recursive positions:

```
data WithRef α β = InR (PF α β)
                 | Ref (Path α)
```

[2] `https://ghc.haskell.org/trac/ghc/wiki/PatternSynonyms`

This is very similar to the metavariable extension for generic rewriting of (Van Noort *et al.*, 2008), only that we extend with *Path* instead of a meta-variable. The type $\mu\,(\mathit{WithRef}\ \alpha)$ is isomorphic to the type $\alpha$ extended with a *Ref* constructor, and thus represents a full tree possibly containing multiple references.

A transformation is then a list of localised insertions of trees with references:

$$\textbf{type}\ \mathit{Transformation}\ \alpha = [\,(\mathit{Path}\ \alpha, \mu\,(\mathit{WithRef}\ \alpha))\,]$$

The *Path* describes the location of the insertion. Note that this *Path* is relative to the earlier edits, and describes a path in the intermediate state of the tree. The *Path*s in the *Ref*s, however, describe a path in the original tree.

### 6.1.1 Pattern synonyms for trees with references

To facilitate the understanding of *Path*s and values with references, we pretty-print them using the constructor names of the original data types, and remove the sum of product structure induced by the `regular` library. The actual representation of the insertion of the *Neg* constructor shown previously is as follows:

$$\begin{aligned}
&\mathit{insert} :: \mathit{Maybe}\ \mathit{Expr}\\
&\mathit{insert} = \mathit{apply}\ \mathit{addNeg}\ \mathit{expr}_1\ \textbf{where}\\
&\quad \mathit{addNeg} :: \mathit{Transformation}\ \mathit{Expr}\\
&\quad \mathit{addNeg} = [\,(\mathit{Add}_1\ \mathit{End}, \mathit{In} \circ \mathit{InR} \circ R \circ R \circ L \circ I \circ \mathit{In}\ \$\ \mathit{Ref}\ (\mathit{Add}_1\ \mathit{End}))\,]
\end{aligned}$$

Fortunately, we can again use pattern synonyms to represent expressions which may contain references, just like we have used them to encode paths in Section 5.1. We define one pattern synonym per data type constructor:

$$\begin{aligned}
&\textbf{pattern}\ \mathit{Var}_E\ \ x\ \ = \mathit{In}\ (\mathit{InR}\ (L\ (K\ x)))\\
&\textbf{pattern}\ \mathit{Const}_E\ x\ = \mathit{In}\ (\mathit{InR}\ (R\ (L\ (K\ x))))\\
&\textbf{pattern}\ \mathit{Neg}_E\ \ x\ \ = \mathit{In}\ (\mathit{InR}\ (R\ (R\ (L\ (I\ x)))))\\
&\textbf{pattern}\ \mathit{Add}_E\ \ x\ y = \mathit{In}\ (\mathit{InR}\ (R\ (R\ (R\ (I\ x :\times: I\ y)))))\\
&\textbf{pattern}\ \mathit{Ref}_E\ \ x\ \ = \mathit{In}\ (\mathit{Ref}\ x)
\end{aligned}$$

We omit the type signatures for these patterns for brevity (and they can be inferred by the compiler). We also introduce $\mathit{Ref}_E$ to hide the fixpoint constructor *In*.

## 6.2 Applying transformations

To apply a transformation, the original tree is taken as a starting value, and the localised insertions are performed one by one to produce a resulting tree:

$$\begin{aligned}
&\mathit{apply} :: \mathit{Transform}\ \alpha \Rightarrow \mathit{Transformation}\ \alpha \to \alpha \to \mathit{Maybe}\ \alpha\\
&\mathit{apply}\ e\ t = \mathit{foldM}\ (\lambda a\ (p, c) \to \mathit{mapP}\ p\ (\lambda_ \to \mathit{resolveRefs}\ t\ c)\ a)\ t\ e
\end{aligned}$$

The inserted value is constructed using function *resolveRefs*, which will be discussed shortly. Function *mapP* takes care of inserting the value at the correct position, as indicated by its path argument.

**Resolving references** We resolve references from a tree by replacing them with values that we look up from another tree, which in the case of *apply* is the original tree:

$$
\begin{aligned}
&\mathit{resolveRefs} :: (\mathit{Transform}\ \alpha, \mathit{Monad}\ \omega) \Rightarrow \alpha \to \mu\ (\mathit{WithRef}\ \alpha) \to \omega\ \alpha \\
&\mathit{resolveRefs}\ r\ (\mathit{In}\ (\mathit{InR}\ a)) = \mathit{liftM}\ \mathit{to}\ (\mathit{fmapM}\ (\mathit{resolveRefs}\ r)\ a) \\
&\mathit{resolveRefs}\ r\ (\mathit{In}\ (\mathit{Ref}\ p)) = \mathit{extract}\ p\ r
\end{aligned}
$$

This function simply recurses over the tree and uses *extract* to find the part of the tree that is reused.

**Extracting children** The *extract* function takes a path and the original tree, and returns the subtree at that location. It uses the generic function *gextract*, which extracts a child given a *Dir*ection:

$$
\begin{aligned}
&\mathit{extract} :: (\mathit{Transform}\ \alpha, \mathit{Monad}\ \omega) \Rightarrow \mathit{Path}\ \alpha \to \alpha \to \omega\ \alpha \\
&\mathit{extract}\ [\,] \quad\ = \mathit{return} \\
&\mathit{extract}\ (p : ps) = \mathit{gextract}\ p\ (\mathit{extract}\ ps) \circ \mathit{from} \\
&\textbf{class}\ \mathit{Extract}\ \phi\ \textbf{where} \\
&\quad \mathit{gextract} :: \mathit{Monad}\ \omega \Rightarrow \mathit{Dir}\ \phi \to (\alpha \to \omega\ \alpha) \to \phi\ \alpha \to \omega\ \alpha
\end{aligned}
$$

The instances of *Extract* are unsurprising and can be found in our code bundle.

**Indexed mapping** To update the tree in *apply* we use a map function that restricts its application to a specific part of the tree:

$$
\begin{aligned}
&\mathit{mapP} :: (\mathit{MapP}\ (\mathit{PF}\ \alpha), \mathit{Monad}\ \omega, \mathit{Regular}\ \alpha) \Rightarrow \mathit{Path}\ \alpha \to (\alpha \to \omega\ \alpha) \to \alpha \to \omega\ \alpha \\
&\mathit{mapP}\ [\,] \quad\ f = f \\
&\mathit{mapP}\ (p : ps)\ f = \mathit{liftM}\ \mathit{to} \circ \mathit{gmapP}\ p\ (\mathit{mapP}\ ps\ f) \circ \mathit{from} \\
&\\
&\textbf{class}\ \mathit{MapP}\ \phi\ \textbf{where} \\
&\quad \mathit{gmapP} :: \mathit{Monad}\ \omega \Rightarrow \mathit{Dir}\ \phi \to (\beta \to \omega\ \beta) \to \phi\ \beta \to \omega\ (\phi\ \beta)
\end{aligned}
$$

## 6.3 Generic diff

We can now automatically generate a transformation from one tree into another (a *diff* operation). This $\mathit{diff} :: \alpha \to \alpha \to \mathit{Transformation}\ \alpha$ should obey the following law:

$$\forall a, b.\ \mathit{apply}\ (\mathit{diff}\ a\ b)\ a \equiv \mathit{Just}\ b$$

For any given $a$ and $b$ there are many different ways to transform $a$ into $b$. For example, $b$ can be inserted directly at the top level, replacing $a$; this is a valid transformation, albeit unsatisfactory since all sharing is lost.

In this section we describe a *diff* function that results in maximal sharing, so only values that are not present in $a$ are inserted into $b$. The algorithm recursively builds up a set of insertions that transform $a$ into $b$. As the *diff* function is relatively large,

we present it in a step-wise fashion, "uncovering" parts of its definition as we describe each subcomponent.

Note that the algorithm we describe is not necessarily the best possible in terms of efficiency or usability. The main goal of this section is to illustrate how such an algorithm can be constructed, and to provide an example of how to use our representation of transformations.

**Overview** The algorithm works in a top-down fashion by traversing the origin and target trees from the root towards the children. At each node, the best set of insertions is chosen based on whether the current node matches the target tree, whether parts of the original tree can be reused, and based on the insertions for the children. We now describe each subcomponent of the algorithm.

**Existing children** To maximise sharing, existing parts of the tree should be used whenever possible. The following function gathers all subtrees together with their corresponding locations in the tree:

```
childPaths :: (Regular α, Children (PF α)) ⇒ α → [(α, Path α)]
childPaths a = (a, []) : [(r, n : p) | (c, n) ← children′ (from a)
                                     , (r, p) ← childPaths c]
```

Here we use an extended version of the *children* function shown earlier (Section 4.4), with type $\phi\ \alpha \to [(\alpha, Dir\ \phi)]$, that returns a direction pointing to each child together with the child itself.

In the *diff* function we gather all these subtrees with paths in a list for the original tree:

```
diff :: ∀α.(Transform α) ⇒ α → α → Transformation α
diff a b = ... where
  cps :: [(α, Path α)]
  cps = childPaths a
```

**Base cases** The recursive function that constructs the insertions is called *build*. It takes three parameters: a *Bool* indicating if the current tree has been inserted, the current tree $a'$, and the target tree $b'$. The base cases are implemented as follows:

```
diff a b = build False a b where
  ...
  build :: Bool → α → α → Transformation α
  build False a′ b′ | a′ ≡ b′ = []
  build ins   a′ b′           = case lookup b′ cps of
                                  Just p   → [([], In (Ref p))]
                                  Nothing → ...
```

The trivial base case is when $a'$ and $b'$ are equal, and $a'$ has not just been inserted. In case $a'$ has been inserted, for example because the parent of $a'$ did not exist in the original tree, we continue the search for reuse.

The second base case is when $b'$ is present in the list of subtrees of $a$; in that case, we simply build a *Ref* containing the path to that subtree.

**Shallow equality** In our quest for reuse, we need to be able to check if two trees are equal at least in their first constructor. For this we use *shallow equality*:

```
class SEq φ where
  shallowEq :: φ α → φ α → Bool
```

In case the roots of two trees are equal, they can be left unchanged and we can continue trying to unify their children. This is implemented in the *construct* function:

```
build ins a' b' = ... where
  construct :: Bool → α → Maybe (Transformation α)
  construct ins' c =
    if   shallowEq (from c) (from b')
    then Just ∘ concat ∘ updChildPaths $ zipWith (build ins') (children c) (children b')
    else Nothing
```

This function returns a transformation containing the edits for the children, based on some current tree $c$. The function *updChildPaths* extends the *Path*s for all edits with the location of the child being edited.

**Reusing parts of the original tree** In case no subtree of the original tree can be directly reused as a replacement for the full subtree that is being constructed, we try to reuse only the top part of an existing subtree. Using the *construct* function, we recursively create a list of insertions that transforms this existing subtree into the target subtree:

```
build ins a' b' = ... where
  ...
  reuses :: Maybe (Transformation α)
  reuses = foldl best Nothing [addRef p (construct False x) | (x,p) ← childPaths]
    where addRef p = fmap (λx → ([],In (Ref p)) : x)
```

Since there might be several valid possibilities, we use a function *best* to pick the "best" transformation. The definition of what the best transformation is varies from application to application; in our implementation, we have chosen to return the transformation with the fewest insertions.

**Insertion** When no existing parts of the tree can be reused, we're forced to insert. This insertion is again a tree with references, so we recursively continue constructing insertions that reuse existing parts:

```
build ins a' b' = ... where
  ...
  insert :: Transformation α
```

```
insert = ([], r') : e' where
  Just r  = construct True b'
  (r', e') = partialApply (withRef b') r
```

Function *withRef* lifts a regular tree to a tree with references (never introducing the *Ref* constructor). Insertion can never fail, so we do not return a *Maybe* here.

Initially, an insertion inserts the full target subtree. However, in order to maximise sharing, we recursively try to replace parts of this target subtree by parts coming from the original tree, using references. To make the inserted value as small as possible, we directly apply these insertions to the inserted tree using *partialApply*.

**Completing the diff** Finally, we combine the previous definitions to construct the return value for the diff. The preferred return value is the case where a value is reused, and only if no values can be reused is the insertion returned:

```
diff a b = build False a b where
  ...
  build ins a' b' =
    case lookup b' cps of
      ...
      Nothing → maybe insert id uses where
        uses :: Maybe (Transformation α)
        uses = if ins then reuses <|> construct ins a'
                      else reuses `best` construct ins a'
```

We use the $(<|>)$ operator as a left-biased choice for *Maybe* values.

As an example, let us look at the rebalancing of binary trees again, which can be described as follows:

```
Add a (Add b c) ⇝ Add (Add a b) c
```

When we apply our *diff* function to these trees for some $a$, $b$, and $c$, we obtain the following transformation:

```
[(End,              Ref (Add1 End))
,(Add0 End,         Ref End)
,(Add0 Add1 End, Ref (Add1 (Add0 End)))]
```

Applying these three insertions one-by-one gives us the following transformation steps:

```
   Add a (Add b c)
⇝ Add b c
⇝ Add (Add a (Add b c)) c
⇝ Add (Add a b) c
```

**Efficiency** The diff algorithm as presented in this section has an exponential running time, which is not very useful in practice. However, the arguments to *build* are always subtrees of $a$ or $b$, so *memoisation* can be used to store the results of *build* for repeated calls. If $a$ and $b$ both have at most $n$ nodes (and thus $n$ subtrees), then the running time of the algorithm with memoisation becomes $O(n^3)$. We have implemented the memoised variant of *diff*: it can be found in the companion Hackage package.

# 7 Generalising to families of data types

We have presented our approach using the `regular` library for generic programming. However, this imposes the significant restriction of only supporting single data types. We have also developed a solution using the `multirec` library for generic programming, which allows us to support families of mutually-recursive data types. In this section we describe some of the modifications required for representing transformations over families of data types.

## 7.1 Running example

In `multirec`, the basic unit of generic representation is the *family*. We represent families as a type constructor $\phi :: \star \to \star$. Families are *indexed*, each index being one data type in the family. We represent indices using the type variable $\iota :: \star$.

This is easier to understand in a concrete setting, so let us introduce a small family of data types, which we will use as a running example in this section. We extend the *Expr* data type of arithmetic expressions shown before with statements (*Stmt*) and Boolean expressions (*BExpr*):[3]

```
type AExpr = Expr
data BExpr = BConst Bool
           | Not BExpr
           | And BExpr BExpr
           | GT AExpr AExpr
data Stmt  = Seq [Stmt]
           | Assign String AExpr
           | If BExpr Stmt Stmt
           | While BExpr Stmt
           | Skip
```

Together, these three types form a family of data types. To use them in `multirec`, we define a GADT that allows identifying each of the indices in this family:

```
data AST (ι :: ⋆) where
  BExpr :: AST BExpr
```

[3] Adapted from `http://www.haskell.org/haskellwiki/Parsing_a_simple_imperative_language`.

$$\begin{aligned} &\mathit{AExpr} :: \mathit{AST}\ \mathit{AExpr} \\ &\mathit{Stmt}\ \ :: \mathit{AST}\ \mathit{Stmt} \end{aligned}$$

## 7.2 Directions and paths

Like in `regular`, directions are represented as one-hole contexts with no elements in the recursive positions:

$$\begin{aligned} &\textbf{type}\ \mathit{Dir}\ \phi\ \tau\ \iota = \mathit{Ctx}\ \phi\ \tau\ (K_0\ ())\ \iota \\ &\textbf{data family}\ \mathit{Ctx}\ (\psi :: (\star \to \star) \to \star \to \star)\ (\tau :: \star)\ (\rho :: \star \to \star)\ (\iota :: \star) \end{aligned}$$

The data family $\mathit{Ctx}$ is a type-indexed data type that encodes the one-hole contexts for `multirec` pattern functors. It is similar to the `regular` case, and described in detail by Rodriguez Yakushev *et al.* (2009).

We will generally have expressions with many type variables, so we try to name them consistently as follows:

$\phi$ The family of types;

$\tau$ The type of the "top" expression, from where all paths are looked up;

$\rho$ Recursive positions;

$\iota$ One type in the family.

As such, $\mathit{Dir}\ \mathit{AST}\ \mathit{Stmt}\ \mathit{BExpr}$ encodes a direction on a $\mathit{Stmt}$ that points to a $\mathit{BExpr}$. Since we instantiate the $\rho$ of $\mathit{Ctx}$ to $K_0\ ()$, recursive positions will contain simply $K_0\ ()$ constants.

Paths are lists of directions. Unlike in the `regular` case, we cannot use simple lists, because a path in a family has a top-level type and a hole type. Therefore, if we have two paths $p_1 :: \mathit{Path}\ \phi\ \tau\ \iota$ and $p_2 :: \mathit{Path}\ \phi\ \tau'\ \iota'$, we must ensure that $\iota \equiv \tau'$, so that we can compose the two paths. This equality is ensured through a GADT:

$$\begin{aligned} &\textbf{data}\ \mathit{Path}\ (\phi :: \star \to \star)\ (\tau :: \star)\ (\iota :: \star)\ \textbf{where} \\ &\quad \mathit{Empty} :: \qquad\qquad\qquad\qquad\qquad \mathit{Path}\ \phi\ \iota\ \iota \\ &\quad \mathit{Push}\ \ :: \phi\ o \to \mathit{Path}\ \phi\ \tau\ o \to \mathit{Dir}\ (\mathit{PF}\ \phi)\ o\ \iota \to \mathit{Path}\ \phi\ \tau\ \iota \end{aligned}$$

When we $\mathit{Push}$ a new direction onto a path, that direction must start at the type where the previous path ended, and the resulting path will end at the type where the new direction ends. Directions are computed on the pattern functor of the family type, so we use the $\mathit{PF}$ type family. Because the connecting type index $o$ is existentially-quantified, we pass an extra argument of type $\phi\ o$ to help the type-checker with fixing the type of $o$.

## 7.3 Values with references

As in our `regular` encoding, we work with pattern functors extended with references:

```
data WithRef   φ τ ρ ι = InR (PF φ ρ ι)
                       | Ref (Path φ ι τ)
type HWithRef φ τ    ι = HFix (WithRef φ τ) ι
```

As such, $WithRef\ \phi\ \tau\ \rho\ \iota$ represents either a value of type $\iota$ in the $\phi$ family of types, or a reference to a value of type $\iota$ on a value of type $\tau$ in the same family, with $\rho$ controlling the recursive positions. In $HWithRef\ \phi\ \tau\ \iota$, the recursive positions have been replaced with values with references, so this type encodes expressions of type $\iota$ in the family of types $\phi$ with possible paths that are to be looked up in an expression of type $\tau$.

Back to our example: $HWithRef\ AST\ AExpr\ BExpr$ encodes a value of type $BExpr$ with possible references that are to be looked up in an expression of type $AExpr$. The need to keep track of the type of the expression at the top is evidenced in the type of insertions, which pairs an expression of type $o$ with a path on an expression of type $o$ with a reference of type $\iota$:

```
data Insert φ τ ι where
  Insert :: φ o → Path φ o ι → HWithRef φ τ o → Insert φ τ ι
```

A transformation is just a list of insertions on the top-level value, so the $\iota$ type is instantiated to $\tau$:

```
type Transformation φ τ = [Insert φ τ τ]
```

## 7.4 Applying transformations, diff

The functions for applying a transformation and diffing two terms still have a very similar interface to the `regular` version:

```
apply :: (Transform φ) ⇒ φ τ → Transformation φ τ → τ → Maybe τ
diff  :: (Transform φ) ⇒ φ τ → τ → τ → Transformation φ τ
```

The only difference is the inclusion of a witness of type $\phi\ \tau$. The $Transform$ constraint synonym simply aggregates all the generic functionality that is required by these functions.

## 7.5 Memoisation

As with the `regular` implementation, the `multirec` approach makes use of memoisation in order to avoid repeating computations. However, memoisation in a family of data types is more challenging. Essentially, we need a memo table for each type in the family. We accomplish this with a type-level list of maps from pairs of values to their difference (a list of insertions):

```
type family MemoTable (φ :: ⋆ → ⋆) (τ :: ⋆) (ι′ :: [⋆]) :: [⋆]
type instance MemoTable φ τ []       = []
type instance MemoTable φ τ (ι : ι′) = Map (MemKey ι) (MemVal φ τ ι)
                                     : MemoTable φ τ ι′
type MemKey      ι = (ι, ι)
type MemVal φ τ ι = [Insert φ τ ι]
```

Here the $\iota'$ parameter is a type-level list containing the indices of all types in the family. Using the standard utilities for manipulating a heterogeneous collection (Kiselyov *et al.*, 2004), insertion and lookup in our memo table proceed first inductively on the heterogeneous collection, and then regularly on the map itself.

The type-level list of indices is given by the user, by instantiating a type family *Ixs*. We show this family together with the instance for our running example:

```
type family Ixs (φ :: ⋆ → ⋆) :: [⋆]
type instance Ixs AST = [AExpr, BExpr, Stmt]
```

Instances for *Ixs* could be automatically generated, but by leaving them to be defined by the user we allow manual control over which types get memoised: indices of the family that are not present in this list will be treated as constants.

## 7.6 Comparison with previous implementation

In our previous approach, we used the following data type to encode values of potentially different types within a same family:

```
data Any φ where
  Any :: Eq_S φ ⇒ φ ι → ι → Any φ
```

Two important places where the *Any* type was used are the list of all children, which can be of different types, and the memoisation tables. Each time that a lookup was performed, for example to find out whether a certain subtree exists as a child, for each element in the list the types needed to be checked for equality through the $Eq_S$ class. Hence, time was spent for lookup even for elements of the wrong type.

Although this approach is flexible, it cluttered up the code and hurt performance. In our new encoding, we do not use the *Any* type, nor values of existentially-quantified index types. Instead, the types are fixed at compile time and the lookup functions only need to look at elements of the right type.

To confirm that the performance of the new approach is better, we benchmarked the old and new approaches using `criterion`.[4] Our benchmark generates 40 pairs of random binary trees, computes the diff between the two trees, and checks that applying the resulting diff to one tree yields the other. We obtained running times of 46.2*ms* for the old approach vs. 41.7*ms* for the new approach, a decrease of 10% in the running time. Because this example contains only a single type, the decrease in runtime only comes from avoiding unneccessary runtime checks. In a family with multiple data types there is more to gain, as we show in Section 8.2.

[4] `http://www.serpentine.com/criterion/`

# 8 Examples

In this section we show three larger examples that showcase our representation of transformations.

## 8.1 A small language of expressions

Let us consider two programs written in the *AST* language introduced in the previous section. Assume the existence of a function $parseString :: String \to Stmt$, and consider the following two programs:

```
prog1 :: Stmt
prog1 = parseString $
      "a := 1;"
  ++ "b := a + 2;"
  ++ "if b > 3"
  ++ "then a := 2"
  ++ "else b := 1"
```

```
prog2 :: Stmt
prog2 = parseString $
      "a := 1;"
  ++ "b := a + 2;"
  ++ "if not (b > 3)"
  ++ "then b := 1"
  ++ "else a := 2"
```

They differ in the condition of the *If* statement, which is negated in $prog_2$, and the swapping of the actions in the "then" and "else" branches. To demonstrate our *diff*, we observe that the expression *diff Stmt* $prog_1$ $prog_2$ evaluates to:

$$
\begin{array}{l}
[\mathit{Insert\ BExpr}\ (\mathit{Seq}_0\ (\mathit{List}_2\ (\mathit{If}_0\ \mathit{End})))\ (\mathit{Not}_E\ (\mathit{Ref}_E\ (\mathit{Seq}_0\ (\mathit{List}_2\ (\mathit{If}_0\ \mathit{End})))))\\
,\mathit{Insert\ Stmt}\ \ (\mathit{Seq}_0\ (\mathit{List}_2\ (\mathit{If}_1\ \mathit{End})))\ (\mathit{Ref}_E\ (\mathit{Seq}_0\ (\mathit{List}_2\ (\mathit{If}_2\ \mathit{End}))))\\
,\mathit{Insert\ Stmt}\ \ (\mathit{Seq}_0\ (\mathit{List}_2\ (\mathit{If}_2\ \mathit{End})))\ (\mathit{Ref}_E\ (\mathit{Seq}_0\ (\mathit{List}_2\ (\mathit{If}_1\ \mathit{End}))))]
\end{array}
$$

We are using pattern synonyms for paths and constructors extended with references named as before.[5] Our diff performs a perfect job at maintaining sharing: the $Not_E$ is inserted reusing the condition, and the clauses are swapped by inserting $Ref_E$s pointing to the original expression.

## 8.2 Edits in a Lua project

As a larger and more realistic example, we looked at 90 edits made to a Lua source code file in an open source project.[6] The Lapis project was chosen because it was open-source and active; the specific file chosen is reasonably large (733 lines in the final version) and the commits are representative of typical software project edits. The text-based diffs between these versions consist of 18.6 lines on average, with a maximum of 106. Both our old and new versions computed the same transformation for each commit, with the transformations consisting of 8.7 insertions on average with a

[5] The $List_2$ pattern synonym requires further explanation. *Seq*uences are lists of statements. Lists, and other container types, are handled in a adhoc fashion through a composition representation type in `multirec`. Paths on lists are then defined to traverse through the $n$-th element of the list, and this is encoded with the pattern synonym $List_n$.

[6] The application.lua file from the Lapis web framework (`http://leafo.net/lapis/`), from commits 35046ff to 629c559.

maximum of 115 insertions. The new version took 62*s* in total, while the old one took 170*s* to compute all transformations, which is a 63.5% decrease in runtime.

This impressive performance gain highlights the importance of avoiding existentially quantified types such as the *Any* type. Not only can the runtime type comparisons be avoided, also the lookup in the memoisation tables are more efficient in our new version. In the old version the values for all different types in the family were stored in one big table, while in the new version there is a table per type. During *compile time* the right table is chosen, such that the code performing the lookup at runtime only works with values of the right type. In case of the Lua project which consists of 12 different types this improvement therefore leads to large efficiency increase.

### 8.3 Generic storage

In the third example we apply both our approach and a traditional diff to transform binary trees stored in a file on disk. Visser & Löh (2010) describe how to store functional data structures in a heap structure on disk. For each constructor, space is allocated in the heap storing the constructor and a pointer for each of its children. When the tree is inspected only the part of the tree that is inspected needs to be read from disk.

We have implemented a benchmark transforming a binary tree with 39 stored to disk, consisting of the following steps: swap the left and right subtree (both consisting of 19 nodes), store the result to the disk, swap the subtrees back, and finally save the file again. With a traditional diff representation, where one of the subtrees is newly inserted because of the swap, this process takes 35.7*ms*. Using our representation of transformations we can perform this operation in 20.9*ms*.

## 9 Conclusion

In this paper we have highlighted the importance of a good representation of transformations. We have seen many examples of transformations and applications that require keeping track of changes, and have shown an implementation for dealing with this problem. We now review related work, and discuss some shortcomings of our approach, together with possible directions for future work.

### 9.1 Related work

The most closely related work to ours is that of Lempsink *et al.* (2009). They describe how to define a generic, type-safe diff algorithm that operates on families of data types. Their notion of "transformation" is encoded by an edit script, which contains insertion, deletion, and copy operations only. They also define an associated *patch* function that transforms a value according to an edit script. However, as we mentioned previously, our work goes beyond the notion of diff.

The ATerm library (Van den Brand & Klint, 2007) provides a representation for the creation and exchange of tree-like data structures in an untyped setting. The implementation is based on maximal subterm sharing by representing terms as a directed acyclic graph.

The technique of extending pattern functors for supporting additional functionality is commonplace. We have used zippers and references in this work; other applications include selections of subexpressions (Van Steenbergen *et al.*, 2010) and generic storage (Visser & Löh, 2010), for example.

Approaches to datatype-generic rewriting such as (Van Noort *et al.*, 2008) and (Jansson & Jeuring, 2000) may potentially benefit from our approach by using our approach for representing the rewrite rules.

We have implemented a simple diff algorithm using our representation for transformations, but in general there are many more tree diff algorithms. The survey paper (Bille, 2005) lists many of them.

## 9.2 Shortcomings

While our solution provides a good basis for an efficient representation of transformations, it has some potential limitations and shortcomings.

**Type safety** Our approach is type-safe in the sense that it does not go wrong at runtime. We cannot encode invalid paths (such as a path going through the second child of a constructor with only one child).[7] However, a given path might not make sense for a given value; we can encode the transformation that traverses the first child of a constructor, while applying it to a value built with a different constructor. Our *diff* does not return invalid transformations. Our *apply*, when applied to such transformations, fails gracefully at runtime (in the *Maybe* monad).

However, we could aim higher, and try to check validity of transformations already at compile-time. This would require a significantly more complicated approach, and certainly some form of dependent types. We expect that aiming for more type-safety will be an interesting adventure in a dependently-typed approach to representing transformations.

**Error handling** Currently, our approach handles failure by returning *Nothing*. While this is preferable to runtime failure, it is not very informative. An easy way to improve the usability of our system would be to provide more useful feedback in case of failure, such as a *String* detailing what went wrong, and where.

## 9.3 Future work

The most natural step for evaluating the usefulness of our system would be to use it directly in one of the applications we suggest. In fact, this work was motivated by the lack of an appropriate representation of transformations for the implementation of incremental evaluation of attribute grammars (Bransen *et al.*, 2015). As such, we plan to integrate our description of transformations in the Utrecht University Attribute Grammar system (UUAG, Swierstra *et al.*, 1999), and see if they can be put to good use in improving the performance of attribute grammars.

[7] This is an improvement over our previous solution (Bransen & Magalhães, 2013), where paths were simply lists of integers.

However, if improving performance is our goal, we have to pay close attention to the performance of *diff* itself. As mentioned in Section 6.3, its complexity, with memoisation, is $O(n^3)$. Cubic behaviour might still be unacceptable in practical scenarios, but lowering this bound would require trading preservation of reuse for speed. It remains to see where the balance between these two factors lays.

Another way to improve performance is to optimise the handling of transformations with many paths sharing some common prefix. The representation could be extended to share such common prefixes so as to better support localised insertions.

We have not reasoned about transformations, which is another interesting direction for future work. For example, we may want to compose multiple transformations or prove that two given transformations commute.

Our notion of transformations makes sharing between the old and new version of a term explicit by representing in which way the old version can be transformed into the new version. In term graphs (Barendregt *et al.*, 1987) different parts of the input are shared, and it would be interesting to discover in what way our approach can be applied to term graphs such that both types of sharing are represented.

## Acknowledgments

The first author is funded by the research programme "Non-Invasive Incremental Evaluation" of the Netherlands Organisation for Scientific Research (NWO). We thank Patrik Jansson and the anonymous reviewers for the helpful feedback.

# References

Acar, Umut A. (2005). *Self-adjusting computation*. Ph.D. thesis, Carnegie Mellon University.

Barendregt, H.P., van Eekelen, M.C.J.D., Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., & Sleep, M.R. (1987). Term graph rewriting. *Pages 141–158 of:* de Bakker, J.W., Nijman, A.J., & Treleaven, P.C. (eds), *PARLE Parallel Architectures and Languages Europe*. Lecture Notes in Computer Science, vol. 259. Springer Berlin Heidelberg.

Bille, Philip. (2005). A survey on tree edit distance and related problems. *Theoretical Computer Science*, **337**(1-3), 217–239.

Van den Brand, Mark G.J., & Klint, Paul. (2007). ATerms for manipulation and exchange of structured data: It's all about sharing. *Information and Software Technology*, **49**(1), 55–64.

Bransen, Jeroen, & Magalhães, José Pedro. (2013). Generic representations of tree transformations. *Pages 73–84 of: Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*. WGP '13. ACM.

Bransen, Jeroen, Dijkstra, Atze, & Swierstra, S. Doaitse. (2015). Incremental evaluation of higher order attributes. *Pages 39–48 of: Proceedings of the ACM SIGPLAN workshop on Partial Evaluation and Program Manipulation*. PEPM '15. ACM.

Gerdes, Alex. (2012). *Ask-Elle: a Haskell tutor*. Ph.D. thesis, Universiteit Utrecht.

Gibbons, Jeremy. (2007). Datatype-generic programming. Backhouse, Roland, Gibbons, Jeremy, Hinze, Ralf, & Jeuring, Johan (eds), *Spring school on datatype-generic programming*. Lecture Notes in Computer Science, vol. 4719. Springer-Verlag.

Gibbons, Jeremy. (2013). Accumulating attributes (for Doaitse Swierstra, on his retirement). *Pages 87–102 of:* Hage, Jurriaan, & Dijkstra, Atze (eds), *Een lawine van ontwortelde bomen: Liber amicorum voor Doaitse Swierstra*.

Hinze, Ralf, Jeuring, Johan, & Löh, Andres. (2002). Type-indexed data types. *Pages 148–174 of: Proceedings of the 6th International Conference on Mathematics of Program Construction*. Lecture Notes in Computer Science, vol. 2386. Springer-Verlag.

Holdermans, Stefan, Jeuring, Johan, Löh, Andres, & Rodriguez Yakushev, Alexey. (2006). Generic views on data types. *Pages 209–234 of: Proceedings of the 8th International Conference on Mathematics of Program Construction*. Lecture Notes in Computer Science, vol. 4014. Springer.

Jansson, Patrik, & Jeuring, Johan. (2000). A framework for polytypic programming on terms, with an application to rewriting. Utrecht University. UU-CS-2000-19.

Kiselyov, Oleg, Lämmel, Ralf, & Schupke, Keean. (2004). Strongly typed heterogeneous collections. *Pages 96–107 of: Proceedings of the ACM SIGPLAN Workshop on Haskell*. Haskell '04. ACM.

Lempsink, Eelco, Leather, Sean, & Löh, Andres. (2009). Type-safe diff for families of datatypes. *Pages 61–72 of: Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming*. WGP '09. ACM.

Magalhães, José Pedro. (2013). Optimisation of generic programs through inlining. *Pages 104–121 of:* Hinze, Ralf (ed), *Implementation and Application of Functional Languages*. Lecture Notes in Computer Science. Springer Berlin Heidelberg.

McBride, Conor. (2001). *The derivative of a regular type is its type of one-hole contexts*. Unpublished manuscript, available at `http://www.cs.nott.ac.uk/~ctm/diff.pdf`.

Van Noort, Thomas, Rodriguez Yakushev, Alexey, Holdermans, Stefan, Jeuring, Johan, & Heeren, Bastiaan. (2008). A lightweight approach to datatype-generic rewriting. *Pages 13–24 of: Proceedings of the ACM SIGPLAN Workshop on Generic Programming*. WGP '08. ACM.

Peyton Jones, Simon (ed). (2003). *Haskell 98, language and libraries. the revised report*. Cambridge University Press. Journal of Functional Programming Special Issue 13(1).

Reps, Thomas, Teitelbaum, Tim, & Demers, Alan. (1983). Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, **5**(July), 449–477.

Rodriguez Yakushev, Alexey, Holdermans, Stefan, Löh, Andres, & Jeuring, Johan. (2009). Generic programming with fixed points for mutually recursive datatypes. *Pages 233–244 of: Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. ICFP '09. ACM.

Schrage, Martijn M. 2004 (Oct). *Proxima—a presentation-oriented editor for structured documents*. Ph.D. thesis, Universiteit Utrecht.

Van Steenbergen, Martijn, Magalhães, José Pedro, & Jeuring, Johan. (2010). Generic selections of subexpressions. *Pages 37–48 of: Proceedings of the ACM SIGPLAN Workshop on Generic Programming*. WGP '10. ACM.

Swierstra, S. Doaitse, Azero Alcocer, Pablo R., & Saraiva, João. (1999). Designing and implementing combinator languages. *Pages 150–206 of: Advanced functional programming*. Lecture Notes in Computer Science, vol. 1608. Springer Berlin Heidelberg.

Visser, Sebastiaan, & Löh, Andres. (2010). Generic storage in Haskell. *Pages 25–36 of: Proceedings of the ACM SIGPLAN Workshop on Generic Programming*. WGP '10. ACM.