Generic Representations of Tree Transformations

JEROEN BRANSEN
Department of Information and Computing Sciences, Utrecht University

JOSÉ PEDRO MAGALHÃES
Department of Computer Science, University of Oxford

(e-mail: J.Bransen@uu.nl, jpm@cs.ox.ac.uk)

Abstract

Applications which 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, or lose shared structure between the old and new 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 to a specific class of applications: that of 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 datatypes, 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.

Contents

1 Introduction 2
2 Transformation operations 3
  2.1 Localisation 5
  2.2 Diff is not enough 5
3 Generic programming for regular functors 6
  3.1 Representation 6
  3.2 Functoriality of the representation types 7
  3.3 Embedding user-defined types 7
  3.4 Generic functions 8
4 Zippers and paths 9
  4.1 Paths 9
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, as 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 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 datatypes. 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.

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](http://dreixel.net/research/code/transformations.tar.gz). The library itself is available at [http://hackage.haskell.org/package/transformations](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), and which we improved with a well-typed representation of paths based on zippers, pattern synonyms for convenience of usage, and provided a larger example application for.

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; Section 3 provides a brief introduction to generic programming with regular functors, and Section 4 describes how to encode paths in a datatype by using generic one-hole contexts. We proceed by detailing our representation of transformations in Section 5 followed by its generalisation to handle mutually recursive families of datatypes in Section 6. Section 7 provides an application of our approach to tracking transformations between versioned commits to a LUA program. We conclude in Section 8 also discussing possible improvements to our approach.

## 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 datatype:
J. Bransen and J. P. Magalhães

**data** $\text{Expr} = \text{Var String}$

| $\text{Const Int}$ |
| $\text{Neg Expr}$ |
| $\text{Add Expr Expr}$ |

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:

- $\text{expr}_1, \text{expr}_2, \text{expr}_3 :: \text{Expr}$
- $\text{expr}_1 = \text{Add (Const 1) (Var "a")}$
- $\text{expr}_2 = \text{Add (Const 1) (Neg (Var "a"))}$
- $\text{expr}_3 = \text{Add (Var "a") (Const 1)}$

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:

$\text{Var "a"} \Rightarrow \text{Neg (Var "a")}$

The arrow $\Rightarrow$ 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 $\text{Var "a"}$ into the expression $\text{Neg _}$, where the underscore is here used informally to denote a hole in an expression. Our example $\text{expr}_1$ can be transformed into $\text{expr}_2$ using such an insertion, which happens in the context of the second subtree of the full expression.

In our setting, the replacement operation is just insertion at a given location.

**Deletion** A deletion removes part of an expression. For example, $\text{expr}_1$ can be seen as arising from the deletion of the $\text{Neg}$ constructor in $\text{expr}_2$. In reality, however, we see deletion as a form of replacement: $\text{expr}_1$ arises by replacing the $\text{Neg x}$ expression by $x$ in $\text{expr}_2$. 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 $\text{expr}_1$ to $\text{expr}_3$, which has the following shape:

$\text{Add a b} \Rightarrow \text{Add b a}$

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.
Generic Representations of Tree Transformations

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

\[ Add\ a\ (Add\ b\ c) \rightarrow Add\ (Add\ a\ b)\ c \]

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:

\[ Add\ a\ (Const\ 0) \rightarrow Add\ a\ a \]

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:

\[ Neg\ (Neg\ (Neg\ (Neg\ (Neg\ (Neg\ (Neg\ (Const\ 1))))))) \rightarrow Neg\ (Neg\ (Neg\ (Neg\ (Neg\ (Neg\ (Neg\ (Const\ 2))))))) \]

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. Lempinsk et al. (2009) describe a type-safe, datatype-generic diff that may be used to determine changes in terms: given a transformation \(t_1 \rightarrow t_2\), \(\text{diff} t_1 t_2\) 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 \(Add\ (Var\ "b")\ (Var\ "a")\):
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.

3 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 revolve around annotating recursive positions in datatypes. 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 datatypes, or multirec [Rodriguez Yakushev et al., 2009], a generalisation of regular that supports mutually-recursive families of datatypes. 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 6.

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

3.1 Representation

Datatypes are encoded in regular using the following five representation types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>data U</td>
<td>ρ = U</td>
</tr>
<tr>
<td>data I</td>
<td>ρ = I ρ</td>
</tr>
<tr>
<td>data K α</td>
<td>ρ = K α</td>
</tr>
<tr>
<td>data (φ :+: ψ)</td>
<td>ρ = L (φ ρ)</td>
</tr>
<tr>
<td>data (φ ::: ψ)</td>
<td>ρ = φ ρ ::: ψ ρ</td>
</tr>
</tbody>
</table>

Unit, encoded by U, is used for constructors without arguments. Recursive positions, encoded by I, denote occurrences of the datatype 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 :::, are used for constructors with multiple arguments. The regular library also contains representation types for dealing with datatype 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 datatype of Section 2 is encoded in regular as follows:
Generic Representations of Tree Transformations

\[ \text{type } ExprPF = K \text{ String } + : K \text{ Int } + : (I :\times I) \]

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

\[
\text{data } \mu \phi = \text{In } (\phi (\mu \phi))
\]

Now, \( \mu ExprPF \) encodes a datatype that is isomorphic to \( Expr \).

### 3.2 Functoriality of the representation types

The \texttt{regular} library encodes datatypes as functors; the recursive positions are abstracted into a parameter \( \rho \). As such, we can provide \texttt{Functor} instances for the representation types. These are unsurprising, with the action being transported across sums and products, ignored in units and constants, and applied at the recursive positions:

\[
\begin{align*}
\text{instance } \texttt{Functor U} & \text{ where } \\
\text{fmap } \_ U &= U \\
\text{instance } \texttt{Functor } (K \alpha) & \text{ where } \\
\text{fmap } \_ (K x) &= K x \\
\text{instance } \texttt{Functor I} & \text{ where } \\
\text{fmap } f (I r) &= I (f r) \\
\text{instance } (\texttt{Functor } \phi, \texttt{Functor } \psi) & \Rightarrow \texttt{Functor } (\phi +: \psi) \text{ where } \\
\text{fmap } f (L x) &= L (fmap f x) \\
\text{fmap } f (R x) &= R (fmap f x) \\
\text{instance } (\texttt{Functor } \phi, \texttt{Functor } \psi) & \Rightarrow \texttt{Functor } (\phi :\times \psi) \text{ where } \\
\text{fmap } f (x :\times y) &= fmap f x :\times fmap f y
\end{align*}
\]

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

### 3.3 Embedding user-defined types

To provide a convenient interface for generic functions, \texttt{regular} uses a type class to aggregate generic representations of user datatypes. This class defines how to represent each datatype, and how to convert to and from its representation:

\[
\begin{align*}
\text{class } \texttt{Regular } \alpha & \text{ where } \\
\text{type } PF \alpha & :: \star \rightarrow \star \\
\text{from } :: & \alpha \rightarrow PF \alpha \alpha \\
\text{to } & :: PF \alpha \alpha \rightarrow \alpha
\end{align*}
\]

The type family \( PF \) encodes the pattern functor of the datatype being represented. The conversion functions \texttt{from} and \texttt{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 datatype 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 $\text{Expr}$ in $\text{regular}$:

```haskell
instance \text{Regular Expr where}
    \text{type PF Expr = Expr}

    \text{from (Var s) = L (K s)}
    \text{from (Const i) = R (L (K i))}
    \text{from (Neg e) = R (R (L (I e)))}
    \text{from (Add e1 e2) = R (R (R (L (I e1) :: I e2)))}

    \text{to (L (K s)) = Var s}
    \text{to (R (L (K i))) = Const i}
    \text{to (R (R (L (I e)))) = Neg e}
    \text{to (R (R (R (L (I e1) :: I e2)))) = Add e1 e2}
```

Instances of the $\text{Regular}$ class are tedious to write by hand; fortunately, the $\text{regular}$ library includes Template Haskell code to automatically generate these instances for user datatypes.

### 3.4 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:

```haskell
class Children \phi where
    gchildren :: \phi \rho \rightarrow [\rho]

instance Children U where
    gchildren _ = []

instance Children I where
    gchildren (I x) = [x]

instance Children (K \alpha) where
    gchildren _ = []

instance (Children \phi, Children \psi) \Rightarrow Children (\phi :+: \psi) where
    gchildren (L x) = gchildren x
    gchildren (R x) = gchildren x

instance (Children \phi, Children \psi) \Rightarrow Children (\phi ::: \psi) where
    gchildren (x ::: y) = gchildren x ++ gchildren y
```

The function $gchildren$ operates on generic representations. We also define the function $\text{children}$ which operates directly on user datatypes, by first converting them to generic representations:

```haskell
children :: (\text{Regular} \alpha, \text{Children} (\text{PF} \alpha)) \Rightarrow \alpha \rightarrow [\alpha]
children = gchildren \circ \text{from}
```

The call $\text{children expr1}$, for example, returns $[\text{Const} 1, \text{Var} "a"]$, as expected. The $\text{to}$ function is not used here because $\text{from}$ leaves the recursive positions unchanged.
4 Zippers and paths

The zipper is a data structure used to represent traversals in a term. It is a type-indexed datatype \cite{Hinze2002}: every algebraic datatype 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; \cite{RodriguezYakushev2009} describe a zipper for families of datatypes.

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 $\text{Loc}$ datatype:

\[
\text{data } \text{Loc } \alpha \text{ where } \\
\text{Loc} :: \text{(Regular } \alpha \text{)} \Rightarrow \alpha \rightarrow \left[ \text{Ctx (PF } \alpha \text{)} \alpha \right] \rightarrow \text{Loc } \alpha
\]

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 datatype \cite{McBride2001}. This type-indexed datatype is encoded in $\text{regular}$ as a data family, indexed over the five representation types:

\[
\text{data family } \text{Ctx } (\phi :: \star \rightarrow \star) :: \star \rightarrow \star \\
\text{data instance } \text{Ctx } U \rho = \text{C}_I \\
\text{data instance } \text{Ctx } (K \alpha) \rho = \text{C}_I \\
\text{data instance } \text{Ctx } (\phi :+; \psi) \rho = \text{C}_L (\text{Ctx } \phi \rho) \\
| \text{C}_R (\text{Ctx } \psi \rho) \\
\text{data instance } \text{Ctx } (\phi :\times; \psi) \rho = \text{C}_1 (\text{Ctx } \phi \rho) (\psi \rho) \\
| \text{C}_2 (\phi \rho) (\text{Ctx } \psi \rho)
\]

Units and constants contain no recursive positions, and as such have an empty context. The $\text{C}_I$ constructor signals a recursive position. Sums can either have a context on the left ($\text{C}_L$) or on the right ($\text{C}_R$). For products, we can choose to traverse the first argument, keeping the second argument intact ($\text{C}_1$), or to do the opposite ($\text{C}_2$).

4.1 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 \cite{Gibbons2013}. Paths on a datatype are then a list of such directions on the corresponding pattern functor:

\[
\text{type } \text{Dir } \phi = \text{Ctx } \phi () \\
\text{type } \text{Path } \alpha = [\text{Dir (PF } \alpha)]
\]

As an example, recall the $\text{Expr}$ type. The following are all the possible navigation directions on this datatype (modulo $\perp$):
J. Bransen and J. P. Magalhães

\[ \text{neg}_0, \text{add}_0, \text{add}_1 :: \text{Dir} \ (PF \ Expr) \]
\[ \text{neg}_0 = CR \ (CL \ CId) \]
\[ \text{add}_0 = CR \ (CR \ (C1 \ CId \ I ())) \]
\[ \text{add}_1 = CR \ (CR \ (C2 \ I ()) \ CId) \]

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, \([\text{add}_1, \text{neg}_0]\) is the path that traverses to Const 2 in the expression \(\text{Add} \ (\text{Const} \ 0) \ (\text{Neg} \ (\text{Const} \ 2))\).

4.1.1 Pattern synonyms for directions

Since directions are type-indexed datatypes defined on pattern functors, they are cumbersome to build, and require knowledge of the particular encoding of the datatype at hand (i.e. its Regular instance). Values like \text{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\(^1\) for this. Pattern synonyms allow us to define shorthands (syonyms) 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 \(Expr\):

\[
\text{pattern End :: Path Expr} \\
\text{pattern End} = [] \\
\text{pattern Neg}_0 :: \text{Path Expr} \rightarrow \text{Path Expr} \\
\text{pattern Neg}_0 x = CR \ (CL \ CId) : x \\
\text{pattern Add}_0 :: \text{Path Expr} \rightarrow \text{Path Expr} \\
\text{pattern Add}_0 x = CR \ (CR \ (C1 \ CId \ I ())) : x \\
\text{pattern Add}_1 :: \text{Path Expr} \rightarrow \text{Path Expr} \\
\text{pattern Add}_1 x = CR \ (CR \ (C2 \ I ()) \ CId) : x
\]

With these pattern synonyms, \(\text{Add}_1 \ (\text{Neg}_0 \ \text{End})\) encodes the same path as \([\text{add}_1, \text{neg}_0]\), but has the advantage that it can be matched against.

5 Transformations with shared subexpressions

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

\(^1\) \url{https://ghc.haskell.org/trac/ghc/wiki/PatternSynonyms}
Generic Representations of Tree Transformations

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 \texttt{Ref} constructor of type \texttt{Path Expr \rightarrow Expr} to the datatype. For now, we abuse notation to introduce the idea informally, deferring the actual implementation to the next subsection. As an example, the transformation from \texttt{expr}_1 to \texttt{expr}_2 is expressed as follows:

\begin{verbatim}
insert :: Maybe Expr
insert = apply addNeg expr_1 where
  addNeg :: Transformation Expr
  addNeg = [(Add_1 End, Neg (Ref (Add_1 End)))]
\end{verbatim}

The first element of the tuple is a path (using the pattern synonyms introduced in Section 4.1) indicating where the transformation takes place. The second element of the tuple is the value to be inserted at this location, in this case a \texttt{Neg} constructor with a reference. \texttt{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 \texttt{Var "a"}).

The transformation from \texttt{expr}_2 to \texttt{expr}_1 by deletion is encoded as follows:

\begin{verbatim}
delete :: Maybe Expr
delete = apply delNeg expr_2 where
  delNeg :: Transformation Expr
  delNeg = [(Add_1 End, Ref (Add_1 (Neg_0 End)))]
\end{verbatim}

Here we first focus on the second child of \texttt{Add}. At that location, we insert a reference that points to a smaller part of the original subtree. This encodes the notion that the \texttt{Neg} constructor that was “in between” is deleted. Here \texttt{Add}_1 \texttt{End} refers to the second child again, which is the \texttt{Neg} constructor, and \texttt{Add}_1 (\texttt{Neg}_0 \texttt{End}) to the first child of the \texttt{Neg} constructor, which is \texttt{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:

\begin{verbatim}
swap :: Maybe Expr
swap = apply swap' expr_1 where
  swap' :: Transformation Expr
  swap' = [(Add_0 End, Ref (Add_1 End))
           , (Add_1 End, Ref (Add_0 End))]
\end{verbatim}

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.
5.1 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 4. 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 8.2).

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

\[
\text{data } \text{WithRef } \alpha \beta = \begin{cases} \text{InR} (\text{PF } \alpha \beta) \\ \text{Ref} (\text{Path } \alpha) \end{cases}
\]

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

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

\[
\text{type } \text{Transformation } \alpha = [(\text{Path } \alpha, \mu (\text{WithRef } \alpha))]
\]

The \( \text{Path} \) describes the location of the insertion. Note that this \( \text{Path} \) is relative to the earlier edits, and describes a path in the intermediate state of the tree. The \( \text{Paths} \) in the \( \text{Refs} \), however, describe a path in the original tree.

5.1.1 Pattern synonyms for trees with references

To facilitate the understanding of \( \text{Paths} \) and values with references, we pretty-print them using the constructor names of the original datatypes, and remove the sum of product structure induced by the \texttt{regular} library. The actual representation of the insertion of the \( \text{Neg} \) constructor shown previously is as follows:

\[
\text{insert} :: \text{Maybe Expr} \\
\text{insert} = \text{apply addNeg expr1 where} \\
\text{addNeg} :: \text{Transformation Expr} \\
\text{addNeg} = [(\text{Add} \_1 \_1, \text{InR} (\text{InR} (\text{InR} (\text{InR} (\text{I} x)))))]
\]

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 4.1. We define one pattern synonym per datatype constructor, postfixed with \( E \) to denote “extended”:

\[
\begin{align*}
\text{pattern } \text{Var}_E & \quad x = \text{In} (\text{InR} (\text{L} (\text{K} x))) \\
\text{pattern } \text{Const}_E & \quad x = \text{In} (\text{InR} (\text{R} (\text{L} (\text{K} x)))) \\
\text{pattern } \text{Neg}_E & \quad x = \text{In} (\text{InR} (\text{R} (\text{L} (\text{I} x)))) \\
\text{pattern } \text{Add}_E & \quad x y = \text{In} (\text{InR} (\text{R} (\text{R} (\text{R} (\text{L} (\text{I} x) :: \text{I} y)))))) \\
\text{pattern } \text{Ref}_E & \quad x = \text{In} (\text{Ref} x)
\end{align*}
\]

We omit the type signatures for these patterns for brevity (and they can be inferred by the compiler). We also introduce \( \text{Ref}_E \) to hide the fixpoint constructor \( \text{In} \).
5.2 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:

\[
\text{apply} :: \text{Transform } \alpha \Rightarrow \text{Transformation } \alpha \rightarrow \alpha \rightarrow \text{Maybe } \alpha
\]

\[
\text{apply } e \ t = \text{foldM} (\lambda \alpha (p, c) \rightarrow \text{mapP} (\text{flip resolveRefs } c) \ p \ a) \ t \ e
\]

The inserted value is constructed using function \text{resolveRefs}, which will be discussed shortly. Function \text{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:

\[
\text{resolveRefs} :: \text{Transform } \alpha \Rightarrow \alpha \rightarrow \mu (\text{WithRef } \alpha) \rightarrow \text{Maybe } \alpha
\]

\[
\text{resolveRefs } r \ (\text{In } (\text{InR } a)) = \text{fmap to } (\text{fmapM} (\text{resolveRefs } r) \ a)
\]

\[
\text{resolveRefs } r \ (\text{In } (\text{Ref } p)) = \text{extract } p \ r
\]

This function simply recurses over the tree and uses \text{extract} to find the part of the tree that is reused.

Extracting children

The \text{extract} function takes a path and the original tree, and returns the subtree at that location. It uses the generic function \text{gextract}, which extracts a child given a \text{Direction}:

\[
\text{extract} :: (\text{Transform } \alpha, \text{Monad } \omega) \Rightarrow \text{Path } \alpha \rightarrow \alpha \rightarrow \omega \alpha
\]

\[
\text{extract } [] = \text{return}
\]

\[
\text{extract } (p : ps) = \text{gextract } (\text{extract } ps) \ p \circ \text{from}
\]

\[
\text{class Extract } \phi \text{ where}
\]

\[
\text{gextract} :: \text{Monad } \omega \Rightarrow (\alpha \rightarrow \omega \alpha) \rightarrow \text{Dir } \phi \rightarrow \phi \alpha \rightarrow \omega \alpha
\]

The instances of \text{Extract} are unsurprising and can be found in our code bundle.

Indexed mapping

To update the tree in \text{apply} we use a map function that restricts its application to a specific part of the tree:

\[
\text{mapP} :: (\text{MapP } (\text{PF } \alpha), \text{Monad } \omega, \text{Regular } \alpha) \Rightarrow (\alpha \rightarrow \omega \alpha) \rightarrow \text{Path } \alpha \rightarrow \alpha \rightarrow \omega \alpha
\]

\[
\text{mapP } f [] = f
\]

\[
\text{mapP } f (p : ps) = \text{liftM } \circ \text{gmapP } (\text{mapP } f \ ps) \ p \circ \text{from}
\]

\[
\text{class MapP } \phi \text{ where}
\]

\[
\text{gmapP} :: \text{Monad } \omega \Rightarrow (\beta \rightarrow \omega \beta) \rightarrow \text{Dir } \phi \rightarrow \phi \beta \rightarrow \omega (\phi \beta)
\]

5.3 Generic diff

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

\[
\forall a, b. \text{apply } (\text{diff } a \ b) \ a \equiv \text{Just } b
\]
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:

\[
\text{childPaths} :: (\text{Regular } \alpha, \text{Children } (PF \alpha)) \Rightarrow \alpha \rightarrow [(\alpha, \text{Path } \alpha)]
\]

\[
\text{childPaths } a = (a, []) : \left[ (r, n : p) \mid (c, n) \leftarrow \text{children}' \left( \text{from } a \right), (r, p) \leftarrow \text{childPaths } c \right]
\]

Here we use an extended version of the \(\text{children}\) function shown earlier (Section 3.4), with type \(\phi \alpha \rightarrow [(\alpha, \text{Dir } \phi)]\), that returns a direction pointing to each child together with the child itself.

In the \(\text{diff}\) function we gather all these subtrees with paths in a list for the original tree:

\[
\text{diff} :: \forall \alpha. (\text{Transform } \alpha) \Rightarrow \alpha \\rightarrow \alpha \\rightarrow \text{Transformation } \alpha
\]

\[
\text{diff } a \ b = \ldots \text{ where }
\]

\[
\text{cps} :: [(\alpha, \text{Path } \alpha)] \\
\text{cps} = \text{childPaths } a
\]

Base cases

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

\[
\text{diff } a \ b = \text{build } \text{False } a \ b \text{ where }
\]

\[
\ldots
\]

\[
\text{build} :: \text{Bool} \rightarrow \alpha \\rightarrow \alpha \\rightarrow \text{Transformation } \alpha
\]

\[
\text{build } \text{False } a' \ b' \mid a' \equiv b' = []
\]

\[
\text{build ins } a' \ b' \rightleftharpoons \text{case lookup } b' \ \text{cps of }
\]

\[
\text{Just } p \rightarrow [([] , \text{In} (\text{Ref } p))]
\]

\[
\text{Nothing} \rightarrow \ldots
\]
Generic Representations of Tree Transformations

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 \( \text{Ref} \) 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*:

```haskell
class SEq \( \phi \) where
    shallowEq :: \( \phi \alpha \rightarrow \phi \alpha \rightarrow \text{Bool} \)
```

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 \( \text{construct} \) function:

```haskell
build ins \( a' b' \) = \ldots \text{where}
    construct :: \( \text{Bool} \rightarrow \alpha \rightarrow \text{Maybe} (\text{Transformation} \alpha) \)
    construct ins' c =
    \quad \text{if} \ shallowEq (from c) (from b')
    \quad \text{then} \ Just \circ \text{concat} \circ \text{updChildPaths} \circ \text{zipWith} (\text{build ins'}) \circ (\text{children c}) \circ (\text{children b'})
    \quad \text{else} \ Nothing
```

This function returns a transformation containing the edits for the children, based on some current tree \( c \). Function \( \text{updChildPaths} \) extends the \( \text{Paths} \) for all edits with the current child indices.

**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 \( \text{construct} \) function, we recursively create a list of insertions that transforms this existing subtree into the target subtree:

```haskell
build ins \( a' b' \) = \ldots \text{where}
    \ldots
    \quad \text{reuses} :: \text{Maybe} (\text{Transformation} \alpha)
    \quad \text{reuses} = \text{foldl} \text{best} \text{Nothing} [\text{addRef} p (\text{construct} \text{False} x) | (x,p) \leftarrow \text{childPaths}]
    \quad \text{where} \text{addRef} p = \text{fmap} (\lambda x \rightarrow ([], \text{In} (\text{Ref} p)) : x)
```

Since there might be several valid possibilities, we use a function \( \text{best} \) 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:

```haskell
build ins \( a' b' \) = \ldots \text{where}
    \ldots
```
**Completion**

**Insertion**

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:

\[
\text{diff } a \ b = \text{build } False \ a \ b \text{ where }
\]

\[
\text{build } \text{ins } a' \ b' =
\]

\[
\text{case } \text{lookup } b' \ \text{cps of}
\]

\[
\text{Nothing } \rightarrow \text{maybe } \text{insert } \text{id} \ \text{uses } \text{where}
\]

\[
\text{uses } : \text{Maybe } (\text{Transformation } \alpha)
\]

\[
\text{uses } = \text{if } \text{ins } \text{then } \text{reuses } < | > \ \text{construct } \text{ins } a'
\]

\[
\text{else } \text{reuses } '\text{best}' \ \text{construct } \text{ins } a'
\]

We use the Swierstra dike 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:

\[
\text{Add } a \ (\text{Add } b \ c) \rightarrow \text{Add } (\text{Add } a \ b) \ c
\]

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

\[
[(\text{End}, \ \text{Ref } (\text{Add} \ 1 \ \text{End}))
 \ (\text{Add} \ 0 \ \text{End}, \ \text{Ref } \text{End})
 \ (\text{Add} \ 0 \ \text{Add} \ 1 \ \text{End}, \ \text{Ref } (\text{Add} \ 1 \ \text{Add} \ 0 \ \text{End}))]
\]

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

\[
\text{Add } a \ (\text{Add } b \ c)
\]

\[
\rightarrow \text{Add } b \ c
\]

\[
\rightarrow \text{Add } (\text{Add } a \ (\text{Add } b \ c)) \ c
\]

\[
\rightarrow \text{Add } (\text{Add } a \ b) \ c
\]

**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 on the companion Hackage package.

6 Generalising to families of datatypes

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

6.1 Running example

In multirec, the basic unit of generic representation is the family. We represent families as a type constructor $\phi :: \star \rightarrow \star$. Families are indexed, each index being one datatype 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 datatypes, which we will use as a running example in this section. We extend the $\text{Expr}$ datatype of arithmetic expressions shown before with statements ($\text{Stmt}$) and Boolean expressions ($\text{BExpr}$):

```haskell
type $AExpr = \text{Expr}$
data $BExpr = \text{BConst Bool}$
    | $\text{Not BExpr}$
    | $\text{And BExpr BExpr}$
    | $\text{GT AExpr AExpr}$
data $Stmt = \text{Seq [Stmt]}$
    | $\text{Assign String AExpr}$
    | $\text{If BExpr Stmt Stmt}$
    | $\text{While BExpr Stmt}$
    | $\text{Skip}$
```

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

```haskell
data $\text{AST ($\iota :: \star$)}$ where
    $\text{BExpr :: AST BExpr}$
    $\text{AExpr :: AST AExpr}$
    $\text{Stmt :: AST Stmt}$
```

2 Adapted from http://www.haskell.org/haskellwiki/Parsing_a_simple_imperative_language
6.2 Directions and paths

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

\[
\text{type } \text{Dir } \phi \tau \iota = \text{Ctx } \phi \tau (K_0()) \iota
\]

\[
\text{data family } \text{Ctx } (\psi :: (\ast \rightarrow \ast) \rightarrow \ast) (\tau :: \ast) (\rho :: \ast \rightarrow \ast) (\iota :: \ast)
\]

The data family \text{Ctx} is a type-indexed datatype that encodes the one-hole contexts for \text{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;
- \(\rho\): Recursive positions;
- \(\tau\): The type of the “top” expression, from where all paths are looked up;
- \(\iota\): One type in the family.

As such, \text{Dir AST Stmt BExpr} encodes a direction on a \text{Stmt} that points to a \text{BExpr}. Since we instantiate the \(\rho\) of \text{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 \(p_1 :: \text{Path } \phi \tau \iota\) and \(p_2 :: \text{Path } \phi \tau' \iota'\), we must ensure that \(\iota \equiv \tau'\), so that we can compose the two paths. This equality is ensured through a GADT:

\[
\text{data } \text{Path } (\phi :: \ast \rightarrow \ast) (\tau :: \ast) (\iota :: \ast) \text{ where }
\]

Empty :: Path \(\phi \iota \iota\)

Push :: \(\phi \circ \rightarrow \text{Path } \phi \tau \circ \rightarrow \text{Dir } (PF \phi) \circ \iota \rightarrow \text{Path } \phi \tau \iota\)

When we \text{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 \(PF\) type family. Because the connecting type index \(\circ\) is existentially-quantified, we pass an extra argument of type \(\phi \circ\) to help the type-checker with fixing the type of \(\circ\).

6.3 Values with references

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

\[
\text{data } \text{WithRef } \phi \tau \rho \iota = \text{InR } (PF \phi \rho \iota) \\
| \text{Ref } (\text{Path } \phi \iota \tau)
\]

\[
\text{type } \text{HWithRef } \phi \tau \iota = \text{HFix } (\text{WithRef } \phi \tau) \iota
\]

As such, \text{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 \text{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 \(\tau\).
Generic Representations of Tree Transformations

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 `ι`:

```
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 `ι` type is instantiated to `τ`:

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

### 6.4 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 → τ → Transformation φ τ → Maybe τ
ndiff :: (Transform φ) ⇒ φ τ → τ → τ → Transformation φ τ
```

The only difference is the inclusion of a witness of type `φ τ`. The `Transform` constraint synonym simply aggregates all the generic functionality that is required by these functions.

### 6.5 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 datatypes 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 τ'
```

```
type MemKey τ = (ι,ι)
type MemVal φ τ ι = [Insert φ τ ι]
```

Here the `τ'` 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 \texttt{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.

6.6 Comparison with previous implementation

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

\begin{verbatim}
data Any φ where
  Any :: EqS φ ⇒ φ ι → ι → Any φ
\end{verbatim}

Although this might seem flexible at first, we ended up having to perform several runtime type comparisons (through the \texttt{EqS} class). This cluttered up the code and hurt performance. In our new encoding, we do not use the \texttt{Any} type, nor values of existentially-quantified index types.

To confirm that the performance of the new approach is better, we benchmarked the old and new approaches using \texttt{criterion}. Our benchmark generates 40 pairs of random 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\text{ms} for the old approach vs. 41.7\text{ms} for the new approach, a decrease of 10% in the running time.

7 Examples

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

7.1 A small language of expressions

Let us consider two programs written in the \texttt{AST} language introduced in the previous section. Assume the existence of a function \texttt{parseString :: String → Stmt}, and consider the following two programs:

\begin{verbatim}
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"
\end{verbatim}

They differ in the condition of the \texttt{if} statement, which is negated in \texttt{prog2}, and the swapping of the actions in the “then” and “else” branches. To demonstrate our \texttt{diff}, we observe that the expression \texttt{diff Stmt prog1 prog2} evaluates to:

\begin{verbatim}
3 http://www.serpentine.com/criterion/
\end{verbatim}
Generic Representations of Tree Transformations

\[
\begin{align*}
&\text{Insert BExpr} \left( \text{Seq}_0 \left( \text{List}_2 \left( \text{If}_0 \text{ End} \right) \right) \right) \left( \text{Not}_E \left( \text{Ref}_E \left( \text{Seq}_0 \left( \text{List}_2 \left( \text{If}_0 \text{ End} \right) \right) \right) \right) \right) \\
&\text{Insert Stmt} \left( \text{Seq}_0 \left( \text{List}_2 \left( \text{If}_1 \text{ End} \right) \right) \right) \left( \text{Ref}_E \left( \text{Seq}_0 \left( \text{List}_2 \left( \text{If}_2 \text{ End} \right) \right) \right) \right) \\
&\text{Insert Stmt} \left( \text{Seq}_0 \left( \text{List}_2 \left( \text{If}_2 \text{ End} \right) \right) \right) \left( \text{Ref}_E \left( \text{Seq}_0 \left( \text{List}_2 \left( \text{If}_1 \text{ End} \right) \right) \right) \right)
\end{align*}
\]

We are using pattern synonyms for paths and constructors extended with references named as before. Our diff performs a perfect job at maintaining sharing: the \text{Not}_E is inserted reusing the condition, and the clauses are swapped by inserting \text{Ref}_E's pointing to the original expression.

7.2 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. The family of datatypes encoding Lua programs consists of 12 datatypes, 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 could use an existing Haskell parser for Lua. 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. Both our old and new versions computed the same edit script for each commit, but the new version did so in 62s, while the old one took 170s. This impressive performance gain highlights the importance of avoiding runtime type comparisons.

8 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.

8.1 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 datatypes. 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.

---

4 The \text{List}_n pattern synonym requires further explanation. Sequences are lists of statements. Lists, and other container types, are handled in an 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 \text{List}_n.

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

8.2 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). 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.

8.3 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.

However, if improving performance is our goal, we have to pay close attention to the performance of diff itself. As mentioned in Section 5.3, its complexity, with memoisation, is $O(n^3)$. Cubic behaviour might still be unacceptable in practical scenarios, but lowering

---

6 This is an improvement over our previous solution [Bransen & Magalhães, 2013], where paths were simply lists of integers.
Generic Representations of Tree Transformations

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.

Acknowledgments

The first author is funded by the research programme "Non-Invasive Incremental Evaluation" of the Netherlands Organisation for Scientific Research (NWO). The second author is funded by EPSRC grant EP/J010995/1.

Bibliography


J. Bransen and J. P. Magalhães


