Generic Representations of Tree Transformations

Jeroen Bransen
Department of Information and Computing Sciences,
Utrecht University
J.Bransen@uu.nl

José Pedro Magalhães
Department of Computer Science,
University of Oxford
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 three different approaches to the problem, of varying complexity, flexibility, and user-friendliness; different approaches might be better suited for one particular application area or another. We provide practical examples of how to encode transformations in each of the approaches, as well as suggestions on how to apply our solution to larger applications. All approaches use the same underlying generic programming library, and support transformations over families of mutually recursive datatypes.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming

Keywords datatype-generic programming; diff; editing; Haskell; transformation

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, 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 (Acar 2005; Reps et al. 1983). An example is the incremental evaluation of attribute grammars using change propagation (Bransen et al. 2013): 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

WGP '13, September 28, 2013, Boston, MA, USA.

ACM 978-1-4503-2389-5/13/09. . . $15.00.
http://dx.doi.org/10.1145/2502488.2502490

(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 three concrete solutions. The three approaches have different interfaces and expressive power, but each is well-suited to a specific kind of application, so we do not consider any of them to be superior to the others. All our code is available online at `http://hackage.haskell.org/package/transformations`.

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 solutions are generic; Section 3 provides a brief introduction to generic programming with regular functors. We proceed by showing different approaches to the problem of representing transformations effectively: Section 4 uses a zipper extended with state, Section 5 uses rewrite rules, and Section 6 gives an explicit representation of transformations. Section 7 describes how all the approaches can be adapted to work for the more general case of mutually recursive families of datatypes. We conclude in Section 8, also discussing possible improvements to our approaches.

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:

```
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:

$$Var\ \texttt{"a"} \rightsquigarrow Neg\ (Var\ \texttt{"a"})$$

The arrow "$\rightsquigarrow$" 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.

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

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:

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

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:

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

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"`):

```
Cpy Add $ Cpy Var $ Ins "b"  $ Ins Var $
Cpy "a" $ Del Var  $ Del "b" $ End
```

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 solutions. As we will see in the coming sections, our solutions 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 solutions 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).

3.1 Representation

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

```
data U          ρ = U
data I          ρ = I ρ
data K α        ρ = K α
data (φ :+: ψ)  ρ = L (φ ρ) | R (ψ ρ)
data (φ :×: ψ)  ρ = φ ρ :×: ψ ρ
```

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 $:\times:$, 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:

```
type Expr_PF =  K String
             :+: K Int
             :+: I
             :+: (I :×: 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:

```
data μ φ = In (φ (μ φ))
```

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

3.2 Functoriality of the representation types

The `regular` library encodes datatypes as *functors*; the recursive positions are abstracted into a parameter ρ. As such, we can provide *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:

```
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.

3.3 Embedding user-defined types

To provide a convenient interface for generic functions, `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:

```
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 datatype 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 datatype at the recursive positions (of type $PF\ \alpha\ \alpha$). 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 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:

```
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 datatypes, 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 "a"]`, as expected. The *to* function is not used here because *from* leaves the recursive positions unchanged.

4. Transformation in a zipper with state

Our first approach to representing transformations is based on a *zipper* (Huet 1997). Transformations require navigation over a term, but also the ability to change terms, and to move things around. Zippers allow navigation and edits, but not relocation of elements, for example. As such, we extend a zipper with *state*, storing the stack of contexts to the current location. Being inside a monad, we can store, or remember, elements previously visited, so that we can reuse them in later stages of the traversal.

As an example, this is how we can encode the transformation of inserting a negation into *expr*$_1$ in this approach:

```
insert :: Maybe Expr
insert = navigateZS expr1 $
  do downZS
     rightZS
     updateZS Neg
```

In this transformation, we "open" *expr*$_1$ for navigation, and simply move down and right into the second child, where we update the term by wrapping *Neg* around it. There is no need to remember values in this transformation; however, the operation that reverses this edit, removing the *Neg*, does require this:

```
delete :: Maybe Expr
delete = navigateZS expr2 $
  do downZS
     rightZS
     r ← downZS
     upZS
     updateZS (const r)
```

In this transformation, we first move to the argument of *Neg* and remember it. Then, we move up to *Neg* itself, and replace the current focus with what we've just stored. This encodes replacing *Neg x* with *x*, without ever having to inspect *x* itself.

We proceed by first briefly introducing the zipper (Section 4.1), and then explaining how to create a state monad out of zipper navigations (Section 4.2).

4.1 The zipper

The zipper is a data structure used to represent traversals in a term. It is a type-indexed datatype (Hinze et al. 2002): every algebraic datatype induces a zipper, generically. A detailed description of zippers is out of the scope of this paper; Rodriguez Yakushev et al. (2009), for example, 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 *Loc* datatype:

```
data Loc α where
  Loc :: (Regular α, Zipper (PF α))
      ⇒ α → [Ctx (PF α) α] → Loc α
data family Ctx (φ :: ⋆ → ⋆) :: ⋆ → ⋆
```

A location is the point currently in focus in the zipper (of type α), 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 (McBride 2001). This type-indexed datatype is encoded in `regular` as a data family, indexed over the five representation types.

The zipper operations can then be defined on the type of contexts:

```
class Functor φ ⇒ Zipper φ where
  cmap       :: (α → β) → Ctx φ α → Ctx φ β
  fill       :: Ctx φ α → α → φ α
  first, last :: φ α → Maybe (α, Ctx φ α)
  next, prev :: Ctx φ α → α → Maybe (α, Ctx φ α)
```

The *cmap* function encodes the functorial map over contexts. Filling the hole in a context with a specific value is the role of *fill*, while *first*, *last*, *next*, and *prev* are primitive navigation operations.

The interface of the zipper consists of operations that operate directly on locations, and not on contexts. These operations let us open a value for navigation, leave navigation, move up, down, left, and right, and obtain or update the value in focus:

```
enter :: (Regular α, Zipper (PF α)) ⇒ α → Loc α
leave :: Loc α → α

up, down, left, right :: Loc α → Maybe (Loc α)

on      :: Loc α → α
updateM :: Monad φ ⇒ (α → φ α) → Loc α → φ (Loc α)
```

These are the operations that we use for encoding transformations.

4.2 Adding state to a zipper

Since we are dealing only with single functors, creating a state monad out of a zipper is very simple: we only need to have the current location as the state. Given that the traversal operations may fail, we make use of the *Maybe* monad to encode failure. We combine these two monads together using a state transformer to create a new monad:

```
type ZipperMonad α β = StateT (Loc α) Maybe β
```

Unfolding the definition of the *StateT* newtype, we see that our *ZipperMonad* α β corresponds to $Loc\ \alpha \to Maybe\ (\beta, Loc\ \alpha)$, correctly encoding traversals that may fail, and produce a result (which, in our case, will always be the current value of focus).

We now need to reimplement the zipper traversal operations in terms of our monad. First we define a function to embed a given movement into our monad:

```
moveZS :: (Loc α → Maybe (Loc α)) → ZipperMonad α α
moveZS m = StateT $ λl → m l >>= λl' → return (on l', l')
```

This simply updates the location in the state to the new location, and returns the element of focus. Moving up, down, left, and right is implemented in terms of *moveZS*:

```
upZS, downZS, leftZS, rightZS :: ZipperMonad α α
upZS    = moveZS up
downZS  = moveZS down
leftZS  = moveZS left
rightZS = moveZS right
```

Next to traversal, we need a way to update the value in focus. We do this using the zipper function *update*, making sure we update the internal location too:

```
updateZS :: (α → α) → ZipperMonad α α
updateZS f = do l ← get
                let Just l' = updateM (Just ∘ f) l
                put l'
                return (on l')
```

Finally, we provide functions to exit the navigation. The first step is to move up to the top, independently of the current location. This is done by unwinding the stack of contexts:

```
topZS :: ZipperMonad α α
topZS = do Loc x l ← get
           case l of
             [] → return x
             _ → do upZS
                    topZS
```

Leaving a zipper is done by evaluating the monad, given an initial location:

```
leaveZS :: Loc α → ZipperMonad α β → Maybe α
leaveZS s m = evalStateT (m >> topZS) s
```

A typical way to use our zipper monad is by providing an initial value and a traversal over it; we define a function for this purpose, which opens the term, and evaluates the monadic computation over it:

```
navigateZS :: (Regular α, Zipper (PF α))
           ⇒ α → ZipperMonad α β → Maybe α
navigateZS x m = leaveZS (enter x) m
```

We have now introduced all the necessary ingredients to work with our zipper monad. As a final example, we show how to encode a swap transformation:

```
swap :: Maybe Expr
swap = navigateZS expr1 $
  do l ← downZS
     r ← rightZS
     updateZS (const l)
     leftZS
     updateZS (const r)
```

We store the left and right children directly as we navigate down and right, respectively. We then update the right child, before moving back to update the left child.

4.3 Summary

We have seen how to encode transformations using a state monad built on a zipper. This first approach is easy to use and simple to encode, albeit the description of transformations can be a bit verbose. We expect this approach to be particularly useful in the realm of structure editors; our navigation operations could be the default operation for pressing the arrow keys, for example, and remembering and restoring elements can be seen as a form of copy-paste.

However, this approach does not provide a convenient way to inspect transformations. Monadic computations are black boxes which cannot be transformed or changed by other computations (they can only be composed).

5. Transformation by rewriting

Our second approach to representing transformations is a combination of two existing techniques:

1. Transformations are localised, so we want to focus our attention on one part of the value. For this we again use a zipper, encoding paths as a stack of one-hole contexts;
2. Once at the location of interest, transformations can be encoded by *rewrite rules* (Van Noort et al. 2008) that describe how an old term is transformed to a new one.

This approach is particularly intuitive and easy to use. As an example, here is the transformation that creates $expr_2$ by inserting a negation around the second child of $expr_1$:

```
insert :: Maybe Expr
insert = apply [(down >=> right, addNeg)] expr1 where
  addNeg :: Rule Expr
  addNeg = rule $ λx → x :~>: Neg x
```

This transformation first traverses $expr_1$ down and right (into the child *Var* `"a"`), and then applies a rewrite rule that wraps the *Neg* constructor around the current expression.

We have already seen how zippers work (Section 4.1). We now give an introduction to generic rewriting (Section 5.1), and then show how to combine these two concepts into a transformation system (Section 5.2).

5.1 Generic rewriting

Generic rewriting allows the definition and application of rewrite rules to values of any *Regular* datatype. A detailed discussion of rewriting is out of the scope of this paper; we refer the reader to Van Noort et al. (2008) for details. Here we summarise the two main aspects of generic rewriting.

Metavariable extension Rewrite rules allow for the usage of metavariables that stand for arbitrary expressions. These metavariables can be seen in our example transformations of Section 2, such as *Add a b* $\rightsquigarrow$ *Add b a*. Here, the *a* and *b* are metavariables, and not regular *Expr* variables (which are encoded with the *Var* constructor). These metavariables can stand for any expression of type *Expr*. To represent these expressions generically, however, we need to extend *Expr* with the concept of metavariable. This is done by providing an alternative at each recursive position of the datatype representation:

```
type Ext φ     = K Metavar :+: φ
type Metavar   = Int
```

```
type Scheme φ     = μ (Ext φ)
type SchemeOf α   = Scheme (PF α)
```

The type synonym *Ext* extends a functor ϕ by allowing a choice to provide a metavariable (here represented by an *Int*) instead. The type *Scheme* ϕ closes the fixed point, giving us a type that may contain metavariables at the recursive positions. The type *SchemeOf* integrates this in the context of *Regular*, operating on a user datatype α, for which the pattern functor *PF* α is chosen as the argument to *Scheme*. *Scheme Expr*, for example, represents the type of expressions extended with metavariables.

Rule specification Having metavariables, we are only lacking a description of rewrite rules. These are defined as two expressions extended with metavariables, one being the left-hand side, and the other the right-hand side of the rule:

```
infix 5 :⇝:
data RuleSpec α = α :⇝: α
type Rule α     = RuleSpec (SchemeOf α)
```

Note that an expression of type *Rule* α is defined in terms of representation types. This means that the rewrite rules are hard to construct, and require knowledge about the representation of datatypes in `regular`. Fortunately, the `rewriting` library provides a convenient way to define rules using a functional notation. We have already seen the rule *addNeg*, but here is an example of a rule encoding a swap:

```
swapExpr :: Rule Expr
swapExpr = rule $ λa b → Add a b :⇝: Add b a
```

Rules can then be applied to expressions using the *rewriteM* function, which takes a rule, an expression, and returns the transformed expression in case of success, or fails within the monad if the rule cannot be applied:

```
rewriteM :: (Rewrite α, Monad φ) ⇒ Rule α → α → φ α
```

5.2 Transformations by rewriting

We're now ready to combine zippers with rewrite rules to describe transformations. We use a zipper to focus on a point of interest in the tree, and a rewrite rule to encode the transformation to be applied there. A path on a zipper is an expression of type *Loc* $\alpha \to$ *Maybe* (*Loc* α), which we pair with a *Rule* to describe transformations:

```
type Transformation α = [(Loc α → Maybe (Loc α), Rule α)]
```

Since transformations might consist of multiple steps, we use a list of paths and rules. To apply a transformation to a value, we first navigate to the point of interest, apply the rewrite rule, and then leave the zipper:

```
apply :: (Regular α, Rewrite α, Zipper (PF α))
      ⇒ Transformation α → α → Maybe α
apply rs = fmap leave ∘ flip (foldM appRule) rs ∘ enter where
  appRule a (l, r) = l a >>= updateM (rewriteM r)
```

We can now show a few more examples of how to use this approach to describe transformations. Previously we have seen how to insert the *Neg* constructor; similarly, we can also remove it:

```
delete :: Maybe Expr
delete = apply [(down>⇒>right, removeNeg)] expr2 where
  removeNeg :: Rule Expr
  removeNeg = rule $ λx → Neg x :⇝: x
```

This transformation removes the *Neg* constructor on the second child of $expr_2$, transforming it into $expr_1$. We use the Kleisli composition operator (>⇒>) to combine paths to a location, as the type of all traversal operators is *Loc* $\alpha \to$ *Maybe* (*Loc* α).

A transformation that swaps the order of elements in an addition is also easy to express:

```
swap :: Maybe Expr
swap = apply [(return, swapExpr)] expr1
```

Here we apply the transformation from Section 5.1 at the top level; *return* is the trivial operation that keeps the focus unchanged.

5.3 Summary

In this section we have seen how to encode transformations by combining a zipper (to focus on a particular subexpression only) and rewrite rules (to apply the transformations in the focus). This approach is very simple to encode, as it relies almost exclusively on the existing machinery for implementing zippers and rewrite rules. The resulting transformations are expressive and very easy to define. We suspect that this approach would be particularly useful in the context of an application such as the exercise assistant; there, each domain typically has a limited and well-defined set of rules, and solving an exercise consists of applying rules at given locations.

There are also drawbacks to this approach, however. The most significant one is that rules (of type *Rule* α) are easy to build (using the *rule* function), but hard to *inspect*. Since rules are implemented using the (extended) pattern functor, they use generic representations, and not the original constructors of the datatype. As an example, here is the expanded version of rule *swap* given above:

```
expandedSwap :: Rule Expr
expandedSwap =
        In (R (R (R (R
          (I (In (L (K 0))) :×: I (In (L (K 1)))))))))
  :⇝: In (R (R (R (R
          (I (In (L (K 1))) :×: I (In (L (K 0)))))))))
```

The expression *In* (*L* (*K* n)) encodes a metavariable n. This encoding is extremely verbose and bears almost no direct resemblance to the original datatype *Expr*. As such, inspecting these transformations, or transforming them automatically, for example, is not an easy task. We have not succeeded at developing a diff algorithm that, given two expressions, returns the necessary rewrite rules to transform one into the other.

Another drawback is that not all sharing is captured in the this representation. For example, in the *swap* example the *Add* constructor is not changed, but this is not made explicit in the internal representation.

6. Explicit encoding of transformations

While the previous two approaches provide the user with a nice interface for describing and applying tree transformations, they do not allow for easy inspection of the transformations. In the context of incremental computations, however, the aim is not only to transform the tree structures themselves, but also to efficiently recompute values based on the changes to those structures. In this section we describe an approach that encodes the transformations explicitly, and thus provides a suitable interface both for generating and inspecting transformations.

Intuitively, the transformations are encoded as a list of localised insertions in which parts of the original tree can be reused. We use the notion of *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 a part of the original tree.

To represent trees with references we add a *Ref* constructor to the datatype. For now, we abuse notation to introduce the idea informally, deferring the actual implementation to the next section. 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 = [([1], Neg (Ref [1]))]
```

The first element of the tuple is a path indicating where the transformation takes place. We encode a path as a list of (0-based) children indices; $[1]$ thus means the second child. The second element of the tuple is the value to be inserted at this location, 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 = [([1], Ref [1,0])]
```

Here we first focus on the right child of the root. 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 $[1]$ refers to the second child again, which is the *Neg* constructor, and $[1,0]$ 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' = [([0], Ref [1]), ([1], Ref [0])]
```

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.

6.1 Representation

The first part of the representation of transformations is the notion of paths in a tree. We represent paths simply as lists of indices:

```
type Path = [Int]
```

For each constructor, the children are numbered from left to right using 0-based indices. Section 8.2 discusses why we are not using zippers to encode paths here.

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

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

This is very similar to the metavariable extension for generic rewriting of Section 5.1, only that here we extend with *Path* instead of *Metavar*. Also we use two named constructors instead of using `:+:`. The type $\mu\,(WithRef\ \alpha)$ is then isomorphic to the type α extended with a *Ref* constructor, and thus represents a full tree possibly containing multiple references.

We represent a transformation as a list of localised insertions of trees with references:

```
type Transformation α = [(Path, μ (WithRef α))]
```

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.

The actual representation of the insertion of the *Neg* constructor shown in the previous section is as follows:

```
insert :: Maybe Expr
insert = apply addNeg expr1 where
  addNeg :: Transformation Expr
  addNeg = [([1], In ∘ InR ∘ R ∘ R ∘ L ∘ I ∘ In $ Ref [1])]
```

As we have shown in Section 3, the value *Neg e* is represented as $R\,(R\,(L\,(I\,e)))$, so in the *addNeg* transformation these constructors are used to build a value representing *Neg* with a reference. This is clearly not a very convenient interface; we address this issue in Section 6.4.

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. This is implemented as follows:

```
apply :: Transform α ⇒ Transformation α → α → Maybe α
apply e t = foldM apply' t e where
  apply' _ ([],    c) = lookupRefs t c
  apply' a (i : is, c) = fmap to ∘ tmapN f ∘ from $ a where
    f j x | i ≡ j      = apply' x (is, c)
          | otherwise = Just x
```

The function $apply'$ is used to apply a single insertion to the intermediate tree. In case of an empty path, the insertion should happen at the current position; the inserted value is constructed by function *lookupRefs*, which will be discussed shortly. In case of a non-empty path, we call $apply'$ recursively on the i-th child using *tmapN*.

Looking up references To look up the references from the tree we pass the original tree as the first parameter to the function *lookupRefs*:

```
lookupRefs :: Transform α ⇒ α → μ (WithRef α) → Maybe α
lookupRefs r (In (InR a)) = fmap to (fmapM (lookupRefs r) a)
lookupRefs r (In (Ref p)) = extract p r
```

This function simply recurses over the tree and uses *extract* to find the part of the tree that is reused. We use *fmap to* because we are returning a *Maybe* value.

Extracting children The *extract* function takes a *Path* and the original tree, and returns the subtree at that location. It uses the generic function *extractN*, which extracts the i-th child of a tree:

```
class ExtractN ϕ where
  extractN :: Int → ϕ α → Maybe α
extract :: (Regular α, ExtractN (PF α))
        ⇒ Path → α → Maybe α
extract p a = foldM (λx i → extractN i (from x)) a p
```

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

Indexed mapping To update the tree in $apply'$ we use a map function that also provides the child index:

```
class MapN φ where
  mapN  :: Int → (Int → α → Maybe β)
        → φ α → Maybe (φ β)
tmapN :: MapN φ ⇒ (Int → α → Maybe β)
                → φ α → Maybe (φ β)
tmapN = mapN 0
```

The first parameter of *mapN* is an accumulating parameter to keep track of which child we are in; it is initialised by *tmapN*.

6.3 Generic diff

Perhaps the biggest advantage of this explicit encoding of transformations is that we can now automatically generate a transformation from one tree into another, commonly known as a *diff* operation. This $diff :: \alpha \to \alpha \to Transformation\ \alpha$ should obey the following law:

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

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 the explicit 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 all subcomponents.

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)]
childPaths a = (a, []) : [ (r, n : p)
                         | (n, c) ← zip [0..] (children a)
                         , (r, p) ← childPaths c ]
```

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

```
diff :: ∀α.(Eq α, Transform α) ⇒ α → α → Transformation α
diff a b = ... where
  cps :: [(α, Path)]
  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 ∘ updateChildPaths $
           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. Function *updateChildPaths* extends the *Path*s 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 *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 (([], In (Ref p)) :)
```

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 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:

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

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

$$[([], Ref\ [1]), ([0], Ref\ []), ([0,1], Ref\ [1,0])]$$

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

$$\begin{aligned} & Add\ a\ (Add\ b\ c) \\ \rightsquigarrow\ & Add\ b\ c \\ \rightsquigarrow\ & Add\ (Add\ a\ (Add\ b\ c))\ c \\ \rightsquigarrow\ & Add\ (Add\ a\ b)\ c \end{aligned}$$

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.

There are many related tree edit distance problems that are NP-complete, and for which therefore no polynomial time algorithms are known. However, in our representation arbitrary subtrees can be reused to achieve maximal sharing, which allows for a polynomial time algorithm like ours.

6.4 Improving the interface

Exposing the internal representation of trees with references is as undesirable as the explicit encoding of rewrite rules of Section 5.3. As such, we add a layer on top of the approach to provide a more convenient interface to the user. A similar approach can be used to improve the interface for the inspection of rewrite rules.

The internal representation $\mu\ (WithRef\ Expr)$ is isomorphic to the following datatype:

```
data Expr_EH = Var_EH   String
             | Const_EH Int
             | Neg_EH   Expr_EH
             | Add_EH   Expr_EH Expr_EH
             | Ref_EH   Path
```

This datatype is just like *Expr*, only with its constructors renamed, and one new contructor $Ref_{EH} :: Path \to Expr_{EH}$ added. $Expr_{EH}$ is much more convenient to use than $\mu\ (WithRef\ Expr)$, so we wish to present this interface to the user instead. We thus define a type class for transforming back and forth between the internal representation and user-facing datatypes like $Expr_{EH}$:

```
class HasRef α where
  type RefRep α
  toRef   :: μ (WithRef α) → RefRep α
  fromRef :: RefRep α      → μ (WithRef α)
```

The instance for *Expr* is similar to its *Regular* instance given in Section 3.3:

```
instance HasRef Expr where
  type RefRep Expr = Expr_EH
  toRef (In (Ref p))          = Ref_EH p
  toRef (In (InR (L (K s))))  = Var_EH s
  ...
```

We omit most of *toRef*, and *fromRef* entirely, as their definition is unsurprising.

Having *RefRep*, we can now define a transformation as follows:

```
type Transformation_EH α = [(Path, RefRep α)]
```

Using $Transformation_{EH}$, we can write the insertion example from the introduction to this section in almost the same way as we did, only now using actual code that works:

```
insert :: Maybe Expr
insert = apply (fromTransformation_EH addNeg) expr_1 where
  addNeg :: Transformation_EH Expr
  addNeg = [([1], Neg_EH (Ref_EH [1]))]
```

Just like the *Regular* instance of *Expr*, the *HasRef* instance for $Expr_{EH}$ can be automatically derived using Template Haskell. Furthermore, the $Expr_{EH}$ datatype itself can also be automatically generated with Template Haskell, so the user only needs to work with different constructor names for building transformations. We have implemented this functionality in our companion Hackage package.

6.5 Summary

In this section we have seen an explicit representation of tree transformations using the notion of trees with references. These references point to parts of the original tree, thereby providing us with a way of expressing sharing.

This approach also allows us to write a diff algorithm that can construct a transformation based on two given trees. Many choices can be made for constructing this algorithm, so the version given here is not necessarily the best choice for all applications. However, our algorithm always tries to return a transformation that maximises the sharing between the original and target tree.

7. Implementation in `multirec`

We have presented all our approaches using the `regular` library for generic programming. However, this imposes the significant restriction of only supporting single datatypes. We have also developed all the approaches using the `multirec` library for generic programming, which allows us to support families of mutually-recursive datatypes. In this section we first describe some of the

modifications required for representing transformations over families of datatypes, and then give an example that makes use of this functionality.

7.1 Implementation changes

Existentials In `multirec`, the basic unit of generic representation is the *family*. We represent families as a type variable $\phi :: \star \to \star$. Families are *indexed*, and each index is one datatype in the family. We represent indices using the type variable $\iota :: \star$. This is a simplification for presentation purposes; the example in Section 7.2 helps to clarify the usage of families in `multirec`.

All three approaches need to represent values of potentially different types, but within a same family. For this we use a datatype *Any* that is parametrised over the family ϕ, and quantifies existentially over the index type ι:

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

This allows us, for example, to return the list of children of a term as a $[Any\ \phi]$, as now children might have different types.

When unpacking elements of type $Any\ \phi$, we often need to use decidable equality, to establish if two elements in the family are of the same type or not. Using functionality provided by `multirec` for decidable comparison of family indices, we can define a function *matchAny* that checks if the type within the *Any* matches the type of the proof given, returning the value if that is the case:

```
matchAny :: Eq_S φ ⇒ φ ι → Any φ → Maybe ι
```

These changes affect the the internal implementation only: the end-user is not exposed to this added complexity.

Transformation in a zipper with state From the user perspective, this approach does not change very significantly. The implementation requires a zipper for `multirec`, but this was defined previously (Rodriguez Yakushev et al. 2009), so we just reuse the existing code. Some functions in the zipper now require passing an additional parameter which fixes the family on which they operate. In particular, the *update* function, that changes the value at the current location, now has to deal with the fact that this value can have any type that belongs to the family.

Transformation by rewriting For this approach, the first step is to define a rewriting library for `multirec`. This is a relatively straightforward development. The only change to the rewriting interface is that *Rule*s are now parametrised over the family and the specific index they operate on. This is visible in the type of the *rewriteM* function:

```
rewriteM :: Rewrite φ ⇒ Rule φ ι → ι → Maybe ι
```

Having a rewriting library for families of datatypes, we can now generalise the concept of transformation. Transformations are still a list of tuples containing the location and the rule to apply, but now we call the tuple a *Step*, as it needs to existentially quantify over the index of the element where the rule is to be applied. Steps are built with the *step* function, which simply takes the path and rule to apply:

```
type Transformation φ ι = [Step φ ι]
step :: El φ ι' ⇒ (Loc φ I0 ι → Maybe (Loc φ I0 ι))
                → Rule φ ι' → Step φ ι
```

Applying these transformations is done by function *apply*, which simply requires an additional parameter to identify the family:

```
apply :: (Zipper φ (PF φ), Rewrite φ)
      ⇒ Transformation φ ι → φ ι → ι → Maybe ι
```

Explicit encoding of transformations In this approach, we also need to introduce the concept of *Step* as a separate datatype, so as to quantify existentially over the index:

```
type Transformation φ = [Step φ]
data Step φ where
  Step :: φ ι → μ (WithRef φ) ι → Path → Step φ
```

The *apply* and *diff* operations remain mostly unchanged, except for the usual addition of a parameter identifying the family:

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

7.2 Examples

As a more realistic example of transformations, we extend the *Expr* datatype of arithmetic expressions shown before with statements (*Stmt*) and Boolean expressions (*BExpr*):[1]

```
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 datatypes. 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
  AExpr :: AST AExpr
  Stmt  :: AST Stmt
```

The following Template Haskell incantation is all that is required now to derive the generic representations of this family in `multirec`:

```
$(deriveAll "AST)
```

Assume now 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. We can express the transformation from $prog_1$ into $prog_2$ in our zipper with state as follows:

```
example_ZS = navigateZS Stmt prog1 $
  do downZS
```

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

```
  rightZS
  rightZS
    -- Swap
  downZS
  l ← rightZS
  r ← rightZS
  updateZS (λp _ → matchAny p l)
  leftZS
  updateZS (λp _ → matchAny p r)
    -- Add negation
  leftZS
  updateZS (λp e → case p of
                     BExpr → Just (Not e)
                     _     → Nothing)
```

This example shows that the `multirec` zipper allows navigation into all types within the family. When updating the current value, however, extra work needs to be done in order to handle the possibility that it might not be of the expected type.

This same transformation can be encoded in the rewrite rules approach as follows:

```
example_RR = apply [step (down>=>right>=>right) swap
                   ,step down addNot] Stmt prog1 where
  swap   :: Rule AST Stmt
  swap   = rule $ λe a b → If e a b :~>: If e b a
  addNot :: Rule AST BExpr
  addNot = rule $ λe → e :~>: Not e
```

There are several things to note here. The rules, *swap* and *addNot*, operate on the same family *AST*, but on different indices. Their type signatures are given here for clarity only; they are not required. Also, this is our first example with more than one step at the same time; the second path, *down*, is relative to the previous position (at the *If*), and not to the root of the tree. An explicit proof is required only when calling *apply*. While this transformation is rather simple already, it can be made even simpler, using only the following rewrite rule instead of *swap*:

```
rule $ λe a b → If e a b :~>: If (Not e) b a
```

Finally, to demonstrate our *diff* operating on the improved interface of the explicit encoding, we observe that the expression $toTransformation_{EH}$ ($diff$ *Stmt* $prog_1$ $prog_2$) evaluates to:

```
[Step_EH BExpr (Not_EH (Ref_BExpr [2,0])) [2,0]
,Step_EH Stmt  (Ref_Stmt  [2,2])          [2,1]
,Step_EH Stmt  (Ref_Stmt  [2,1])          [2,2]]
```

There are now several types of references, and $Step_{EH}$ takes an additional parameter representing the index. Our diff performs a good job at maintaining sharing: the Not_{EH} is inserted reusing the condition, and the clauses are swapped by inserting Ref_{Stmt}s pointing to the original expression.

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 given three different approaches for dealing with this problem. We now review related work, and discuss some shortcomings of our approaches, 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.

The ATerm library (Van den Brand and 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 a rewriting system in this work; other applications include selections of subexpressions (Van Steenbergen et al. 2010) and generic storage (Visser and Löh 2010), for example.

Implementing a zipper as a state monad is a simple exercise.[2] However, we specifically use this for the purpose of describing transformations. Note that our zipper as a state monad is not related to the work of Schrijvers and Oliveira (2011).

8.2 Shortcomings

While our solutions provide a good basis for an efficient representation of transformations, they have some potential limitations and shortcomings.

Type safety All our approaches are type-safe in the sense that they do not go wrong at runtime. Potential sources of failure, such as navigating to non-existent positions (either with a zipper or an invalid list of child indices), or rewrite rules that fail to apply, lead to runtime failures 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: the types of the navigation functions in the zipper, for example, would depend on the type of value they are applied to. This would require a generalisation of the derivative concept; even the dissection of McBride (2008) does not express this concept. The same problem applies to the applicability of rewrite rules. In any case, aiming for more type-safety would probably be an interesting adventure in a dependently-typed approach to representing transformations.

Paths: zippers or lists of integers? In Sections 4 and 5 we have encoded paths in a structure by means of a zipper. In Section 6, however, we have encoded paths as a list of integers, which represent the indices of the children, followed from the root. While it may be argued that zippers are the best way to encode paths, we have used a list of indices in Section 6 simply because they are more convenient in the internal representation: it is easier to write generic functions such as *extractN* and *mapN* that operate on numeric indices than on zipper paths. We also think that this representation of paths is not harder to understand or visualise than a zipper. However, it is easy to provide a zipper-like interface for paths in Section 6, simply by defining paths as follows:

```
data Dir     = Up | Down | Left | Right
type DirPath = [Dir]
```

Writing conversion functions of types *DirPath* → [*Int*] and [*Int*] → *DirPath* is simple.

Error handling Currently, all our approaches handle 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

[2] See, for example, `http://www.haskell.org/haskellwiki/Zipper_monad`.

transformations 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 transformations would be to implement them in one of the applications we suggest. In fact, our work was motivated by the lack of an appropriate representation of transformations for the implementation of incremental evaluation of attribute grammars (Bransen et al. 2013). 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. Since UUAG needs to inspect the transformations in order to decide which attributes need recomputing, our explicit approach (described in Section 6) is the most applicable to this problem.

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 the performance in the explicit representation 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 second author is funded by EPSRC grant EP/J010995/1. Atze Dijkstra, Ruud Koot, Andres Löh, S. Doaitse Swierstra, the members of the "Algebra of Programming" group at the University of Oxford, the members of the "Reading Club for Software Technology" at Utrecht University, and anonymous reviewers provided helpful feedback on earlier versions of this paper.

References

Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Carnegie Mellon University, 2005.

Mark G.J. van den Brand and Paul Klint. ATerms for manipulation and exchange of structured data: Its all about sharing. *Information and Software Technology*, 49(1):55–64, 2007. doi:10.1016/j.infsof.2006.08.009.

Jeroen Bransen, Atze Dijkstra, and S. Doaitse Swierstra. Lazy stateless incremental evaluation machinery for attribute grammars, 2013. Available at `http://www.staff.science.uu.nl/~brans106/lsiag-machinery.pdf`.

Alex Gerdes. *Ask-Elle: a Haskell Tutor*. PhD thesis, Universiteit Utrecht, 2012.

Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007. doi:10.1007/978-3-540-76786-2.

Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *Proceedings of the 6th International Conference on Mathematics of Program Construction*, volume 2386 of *Lecture Notes in Computer Science*, pages 148–174. Springer-Verlag, 2002. doi:10.1007/3-540-45442-X_10.

Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez Yakushev. Generic views on data types. In *Proceedings of the 8th International Conference on Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*, pages 209–234. Springer, 2006. doi:10.1007/11783596_14.

Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997. doi:10.1017/S0956796897002864.

Eelco Lempsink, Sean Leather, and Andres Löh. Type-safe diff for families of datatypes. In *Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming*, WGP '09, pages 61–72. ACM, 2009. doi:10.1145/1596614.1596624.

José Pedro Magalhães. Optimisation of generic programs through inlining, 2013. Accepted for publication at the 24th Symposium on Implementation and Application of Functional Languages (IFL'12).

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

Conor McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 287–295, 2008. doi:10.1145/1328438.1328474.

Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, pages 13–24. ACM, 2008. doi:10.1145/1411318.1411321.

Simon Peyton Jones, editor. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, 2003. doi:10.1017/S0956796803000315. Journal of Functional Programming Special Issue 13(1).

Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst.*, 5:449–477, July 1983. doi:10.1145/2166.357218.

Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 233–244. ACM, 2009. doi:10.1145/1596550.1596585.

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

Tom Schrijvers and Bruno C.d.S. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 32–44. ACM, 2011. doi:10.1145/2034773.2034781.

Martijn van Steenbergen, José Pedro Magalhães, and Johan Jeuring. Generic selections of subexpressions. In *WGP '10: Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, pages 37–48, New York, NY, USA, 2010. ACM. doi:10.1145/1863495.1863501.

S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João Saraiva. Designing and implementing combinator languages. In *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 150–206. Springer Berlin Heidelberg, 1999. doi:10.1007/10704973_4.

Sebastiaan Visser and Andres Löh. Generic storage in Haskell. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, WGP '10, pages 25–36. ACM, 2010. doi:10.1145/1863495.1863500.