Type-Safe Diff for Families of Datatypes

Eelco Lempsink Sean Leather Andres Löh

Department of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

{emlempsi,leather,andres}@cs.uu.nl

Abstract

The UNIX `diff` program finds the difference between two text files using a classic algorithm for determining the longest common subsequence; however, when working with structured input (e.g. program code), we often want to find the difference between tree-like data (e.g. the abstract syntax tree). In a functional programming language such as Haskell, we can represent this data with a family of (mutually recursive) datatypes. In this paper, we describe a functional, datatype-generic implementation of `diff` (and the associated program `patch`). Our approach requires advanced type system features to preserve type safety; therefore, we present the code in Agda, a dependently-typed language well-suited to datatype-generic programming. In order to establish the usefulness of our work, we show that its efficiency can be improved with memoization and that it can also be defined in Haskell.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Language Constructs and Features*]: Data types and structures

General Terms Algorithms, Reliability, Languages

Keywords Dependent types, Datatype-generic programming, Edit distance

1. Introduction

The UNIX `diff` program [Hunt and Mcilroy, 1976] interprets two files as sequences of lines and outputs an edit script of inserted, deleted, and copied lines to represent the difference between the files. Its counterpart, the `patch` program, takes an edit script (usually produced by `diff`) and a file, applies the changes to lines of the file, and generates a new file whose edit distance from the input file is defined by the input edit script.

As an example of `diff` in action, we consider the difference between two files with JSON data. JavaScript Object Notation [Crockford, 2006] is a simple data-interchange language often used for communicating between a web server and a browser. The first file contains a nested array, and in the second file, the same array has been flattened:

```
[ "foo",
  [ "bar",
    "baz" ] ]
```

```
[ "foo",
  "bar",
  "baz" ]
```


WGP'09, August 30, 2009, Edinburgh, Scotland, UK.

The output of running `diff` on these two files is a string patch equivalent to the following sequence of line edits[1]:

```
  Cpy "[ \"foo\","
$ Del "   [ \"bar\","
$ Del "     \"baz\" ] ]"
$ Ins "   \"bar\","
$ Ins "   \"baz\" ]"
```

A typical `diff` output would ignore copied lines (Cpy) and prefix an inserted line (Ins) with a `+` and a deleted line (Del) with a `-`. When reading the `diff` output for program code from a version control system, it can be difficult to see the true, syntactic changes. In the above case, the edit script tells us that we have deleted two lines and inserted two new ones, when we really want to know that we have simply deleted array borders.

We can produce a better result of the JSON file comparison with a syntax-aware diffing tool. Representing the JSON files as values of an abstract syntax, we can determine precisely which parts of the syntax tree have changed. Here is one possible output:

```
  Cpy     'JSArray'
$ Cpy     '(:)'
$ CpyTree
$ Cpy     '(:)'
$ Del     'JSArray'
$ Del     '(:)'
$ CpyTree
$ CpyTree
$ Del     '[]'
$ End
```

At first glance, this edit script appears longer than the previous one; however, in terms of JSON syntax, it is much more precise about the differences. Since we are only concerned with the differences, we focus on the inserts and deletes, disregarding all copies. We arrive at the key changes: deleting a 'JSArray' and a its related list-like structure, '(:)' and '[]', for the nested elements.

We propose a type-safe, structural, `diff`-like library for families of algebraic datatypes (such as the JSON abstract syntax). Our implementation draws from previous work on algorithms for finding the edit distance between two trees and the maximum common embedded subtree. Unlike other structural diff implementations (see Section 2.1), our approach is unique in that it is type-safe and generic in the choice of datatype family. We can generate edit scripts and perform patching entirely on typed abstract syntax trees without the need to resort to a universal datatype in the process.

Overview

Our research does not stand alone. To give an idea of the background needed to get here, in Section 2, we discuss the algorithms forming the foundation of our presentation as well as some key

[1] The symbol $ is the standard Haskell low-precedence function application.

developments towards a diff[2] for tree-structured data. We also give some background on datatype-generic programming, a concept that we use to implement our diff and patch functions for arbitrary datatypes.

The description of our implementation[3] begins in Section 3 with simple diff and patch functions for lists. We present the development in Agda, a dependently typed programming language [Norell, 2007] with a syntax similar to Haskell. Agda is well-suited to modeling datatype-generic programs [Oury and Swierstra, 2008]. It allows us to demonstrate strong type safety guarantees without the distraction of complex type-level features in a language such as Haskell. For a tutorial on Agda, we refer the reader to Norell [2008].

We then traverse from lists to trees. Section 4 provides a diff and a patch that closely mirror the definitions for lists, but it uses a tree representation that is more appropriate for describing datatypes in an untyped way.

After visiting untyped trees, we find that we need a type-safe representation of a datatype to ensure a valid edit script. In Section 5, we describe the representation and generic functions for diff and patch.

In describing the implementations in these sections, we have necessarily optimized for presentation instead of efficiency. We resolve this issue with a discussion of the changes necessary for memoization in Section 6.

Now that we have a complete and efficient implementation, we would like to use our library in larger applications. However, since Agda has not (yet) reached the stage where it is efficient enough for such development, we implemented the same library in Haskell. In Section 7, we highlight the differences in our Haskell diff and patch, pointing out the translations from dependently type aspects in Agda to type-level programming techniques in Haskell.

Wrapping up the paper, we report on some extensions needed for practical matters in Section 8, and in Section 9, we conclude with some reflection and thoughts on future work.

2. Background

Previous research played a key role in guiding us to the current point. Most notably, there is a wealth of information on the algorithms behind edit distance, differential comparison, pattern matching, and inclusion, both on sequences and on trees. In a similar vein, datatype-generic programming has risen to a prominent place in the improvement of software development techniques and libraries. In order to put our work in the appropriate context, we discuss the development of these two threads in the following sections.

2.1 Algorithms for Diff

The algorithm for finding the generic diff between two datatypes is related to several areas of algorithms research. Our problem is a special case of the *edit distance* problem. The edit distance between entities A and B is the minimum number of primitive *edit operations* to transform A into B. A corresponding *edit script* is the sequence of those operations. The best known example of the edit distance problem is the longest common subsequence problem [Hirschberg, 1975, Bergroth et al., 2000]. It was popularized by the UNIX `diff` program, which finds the differences between two text files with a minimal list of edit operations. In this case, the edit operations are inserting and deleting lines of text.

The development of research naturally evolved beyond strings and other sequential data into algorithms for more complex structured data. In particular, the edit distance for trees [Selkow, 1977, Zhang and Shasha, 1989, Bille, 2005] has seen extensive activity. In this case, the edit operations may include inserting, deleting, updating, or copying single nodes or entire subtrees (and the cost of operations may vary). Tree-structured data may be ordered or unordered, labeled or unlabeled, rooted or unrooted. An important use of tree comparison is change detection [Chawathe et al., 1996, Chawathe and Garcia-Molina, 1997] in data such as XML [Peters, 2005] and program syntax [Yang, 1991]. Tree diff is required in order to support syntax-directed version control [Tieleman, 2006].

Our implementation differs from previous work in that we preserve the type safety of the edit script, so that we can transform one value to another without using an untyped intermediate step. In order to do that for arbitrary datatypes, we turn to datatype-generic programming.

2.2 Datatype-Generic Programming

The techniques of *datatype-generic programming* allow a programmer to define functions that work on the structure of datatypes, removing the need to write similar functions for each datatype [Gibbons, 2007]. Common examples of generic functions include equality, parsing, pretty-printing, and ordered comparison. In Haskell, generic functions have been implemented with language extensions [Jansson and Jeuring, 1997, Löh, 2004], though the current trend is to use libraries.

Most generic programming libraries use a *generic view* to represent the structure of a datatype [Holdermans et al., 2006]. The sum of products in, for example, Extensible and Modular Generics for the Masses [Oliveira et al., 2006] allows functions to be defined by induction on the structure. A fixed point representation such as Multirec's [Rodriguez et al., 2009] enables access to the recursive structure of a family (or system) of types.

Defining a generic view in Haskell is closely related to constructing a *universe* in type theory and dependently typed programming. A universe consists of a datatype of codes and an interpretation function that maps codes to types [Benke et al., 2003, Morris, 2007]. Functions can then be defined by induction on the codes.

We construct a universe for generic diff using lists to represent types, constructors, and fields. Unlike other previously reported generic views, this is well-suited to our needs. We discuss this in more detail in Section 5.

The only work (of which we are aware) that comes close to being a generic diff in a functional programming language is from Piponi [2007a,b] on antidiagonals. The antidiagonal is a construct carrying a pair of provably distinct values of the same type. A value of the antidiagonal contains information about the source and the target value and can therefore be considered to be an edit script. However, no effort is made to keep the script minimal or readable by humans.

3. Lists

We present the functions diff and patch that correspond to the UNIX `diff` and `patch` programs. Let us start with the type of diff:

```
diff : List Item → List Item → Diff
```

We use the standard List datatype to represent a sequence. The Item elements in the list are abstract placeholders for anything; we only require that type to support equality.

The resulting Diff type is intended both for both human consumption and as input to a patch function. A Diff value is an edit

[2] We use the term "diff" in several different manners, often in the general sense of an algorithm, but sometimes in a specific reference to an edit script. The meaning should be clear from the context.

[3] The code in this paper has been formatted using lhs2TEX. This allows us some notational freedom. We do not show the complete code for lack of space, but the sources of this paper type-check. For Agda, we disable termination checking and enable the type rule Set : Set. The use of these modifications is not essential for the algorithms presented, but simplifies the presentation.

script: it contains the sequence of edit operations. We define Diff as a datatype of type Set with a constructor for each edit operation.

```
data Diff : Set where
  ins  : Item → Diff → Diff
  del  : Item → Diff → Diff
  cpy  : Item → Diff → Diff
  end : Diff
```

The possible edit operations are inserting an item (ins), deleting an item (del), copying an item (cpy), and concluding the edit script (end). A complete edit script is constructed as a recursive application of Diff constructors.

The above definition of Diff contains sufficient information to fully reconstruct both the original and target lists, but there are a number of other possible variations. If we omit the Item argument from the cpy constructor, we can still construct the target from the source and vice versa. However, if we omit the Item from both cpy and del, we lose the ability to invert the diff. A further option is to merge multiple subsequent cpy applications in order to reduce the size of the Diff. In this paper, we generally use a definition of Diff that contains all information, but we discuss the usefulness of a few variations in Section 8.

Now that we have introduced the type of an edit script, we can implement our functions diff and patch. We start by defining patch, since it is simpler.

3.1 Patching Lists

Given a diff and a value, patch may produce a patched value:

```
patch : Diff → List Item → Maybe (List Item)
```

Patching can fail. For example, a diff may indicate that a non-existing item should be removed. However, while an arbitrary edit script may cause patch to fail, the following property should hold:

```
patch-diff-spec = ∀ xs ys → patch (diff xs ys) xs ≡ just ys
```

The output of diff always generates a result that will reproduce the same target when passed to patch.

The function is straightforward to implement:

```
patch (ins  x d) ys       = (insert x ◇ patch d          ) ys
patch (del  x d) ys       = (           patch d ◇ delete x) ys
patch (cpy x d) ys        = (insert x ◇ patch d ◇ delete x) ys
patch end       []        = just []
patch end       (y :: ys) = nothing
```

Let us look at each of the cases of patch: The case for ins patches the rest of the list and then inserts an item in the beginning.

```
insert : Item → List Item → Maybe (List Item)
insert x ys = just (x :: ys)
```

While insert is not required to return a Maybe here, later versions of this function will need it, so we use this type to unify the definitions. The case for del fails when the input list is empty or when the item to be deleted does not match the item in the list.

```
delete : Item → List Item → Maybe (List Item)
delete x []       = nothing
delete x (y :: ys) = if x == y then just ys else nothing
```

The case for cpy is a combination of both insert and delete. Lastly, the end case only succeeds if the input is empty.

The operator

```
_◇_ : ∀ {A B C} →
      (B → Maybe C) → (A → Maybe B) → (A → Maybe C)
(g ◇ f) x with f x
...       | nothing = nothing
...       | just y  = g y
```

is monadic composition on Maybe. The curly braces in the type signature surround implicit arguments in Agda. We can omit implicit arguments when calling the function as long as Agda is capable of inferring them from the context. The with keyword in Agda evaluates an expression for pattern matching, and we can use ... to imply the repetition of the left-hand side for the with. The underscores in _◇_ are placeholders for the arguments of a function. Agda allows not only infix functions such as _◇_, but also mixfix as we will see later.

3.2 Diffing Lists

More interesting than patch is the implementation of diff. The goal is to provide the shortest possible diff. To this end, we define a cost function that increments for each edit operation

```
cost : Diff → ℕ
cost (ins  _ d) = 1 + cost d
cost (del  _ d) = 1 + cost d
cost (cpy _ d) = 1 + cost d
cost end       = 0
```

and try to produce a value of type Diff fulfilling patch-diff-spec with minimal cost. Other cost functions are of course possible, and we discuss this adaptation in Section 9.

There is a naïve algorithm for diff defined as follows:

```
diff : List Item → List Item → Diff
diff []       []       = end
diff []       (y :: ys) = ins y (diff []  ys)
diff (x :: xs) []       = del x (diff xs [] )
diff (x :: xs) (y :: ys) = if x == y then best₃ else best₂
  where best₂ = del x (diff      xs (y :: ys))
              ⊓ ins y (diff (x :: xs)     ys )
        best₃ = cpy x (diff      xs      ys )
              ⊓ best₂
```

We have four cases to handle. If both lists are empty, the diff is complete. If the source or target list is empty, we construct an insert or delete, respectively. The last case is the most interesting one. If both lists are non-empty, we compare all possible actions (copying is only possible when the items in source and target are equal) and select the best using _⊓_, the minimum of the total preorder on Diff induced by cost.

```
_⊓_ : Diff → Diff → Diff
_⊓_ dx dy = if cost dx ⩽ cost dy then dx else dy
```

The naïve algorithm is inefficient because we have multiple recursive calls at every step. However, there are several ways in which we can improve the efficiency of the algorithm:

- Using a dynamic programming approach, we can share recursive calls as much as possible.
- Given the cost function above, we can simply always copy if copying is possible, because it will never lead to a higher cost. If we choose to insert or delete an element when copying is possible, we might need to compensate with a delete or insert later, resulting in a higher cost.
- Instead of recomputing the cost of the patches at every recursive step, we can pair the cost computation with the computation of the diff itself. If the cost comparison is lazily evaluated, we can also save on computation of the diff.

The diff problem is strongly related to the problem of finding the longest common subsequence of two lists. Given the resulting Diff, we can extract the longest common subsequence by looking only at the cpy constructors. Similarly, when given the longest common subsequence, we can easily extend it to a Diff by using the next element in the common subsequence to resolve the critical choice between ins and del.

4. Trees

Before we jump into the generic diff for families of datatypes, we first look at a diff algorithm for (labeled, ordered) trees.

```
data Tree : Set where
  node : Label → List Tree → Tree
```

Sometimes called a rose tree, this structure allows for each node to have an arbitrarily-sized forest of subtrees. Much like the Item in the list diff, the definition of Label is not important to the problem; we only require an implementation of _≟_ for labels.

We introduce a diff algorithm adapted from the works of Klein [1998] and Lozano and Valiente [2004]; however, our presentation is tailored so that the relationships to both the list diff above and the forthcoming generic diff are evident. The cited algorithms work with untyped, ordered trees. The Tree type is untyped in the sense that the labels may contain anything and there are no (typed) constraints on children or siblings. In Section 4.3, we look at how untyped trees can serve as a representation for generic diffing and why this is undesirable.

The Diff datatype for trees is nearly the same as that for lists.

```
data Diff : Set where
  ins  : Label × ℕ → Diff → Diff
  del  : Label × ℕ → Diff → Diff
  cpy  : Label × ℕ → Diff → Diff
  end : Diff
```

Along with the node's Label in each constructor, we pair a number representing its arity, the length of the list of children for each node. The need for storing the arity will become clear in the following description of patching.

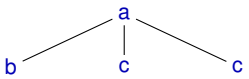
4.1 Patching Trees

For the patch and diff functions, we do not give implementations that work directly with Tree arguments as one might expect. Instead, we use lists of trees. The reason is that in intermediate stages of both patch and diff, situations where we have to deal with multiple subtrees arise naturally. (If desired, one could easily write a wrapper for each function to hide the use of lists.)

To see why lists of trees are in general required, consider

```
node a (node b [] :: node c [] :: node c [] :: [])
```

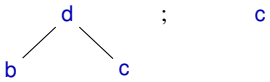
that uses abstract labels a, b, and c. We can depict the tree as



Applying the partial diff del (a,3) to this tree (which effectively removes the node labeled a) will no longer leave a single tree. Instead, we end up with a list of three trees:

```
b  ;  c  ;  c
```

From this point, adding another label by applying the partial diff ins (d,2) reduces the number of trees to two:



As a side note, observe how the arity associated with the label in each edit operation indicates the number of subtrees exchanged for one node. Specifically, the application of del (a,3) produces three new trees from one node in the source list while ins (d,2) consumes two trees to form a new one.

With the idea in mind of lists of trees whose length varies as we insert or delete nodes, we can now implement the patch function. When compared to the patch for lists, its type signature changes to

```
patch : Diff → List Tree → Maybe (List Tree)
```

but its definition remains exactly the same. We only need to reimplement the insert and delete functions:

```
insert : Label × ℕ → List Tree → Maybe (List Tree)
insert (x,n) yss with splitAt n yss
...              | (ys,yss′)    = if length ys ≟ℕ n
                                  then just (node x ys :: yss′)
                                  else nothing

delete : Label × ℕ → List Tree → Maybe (List Tree)
delete (x,n) []                 = nothing
delete (x,n) (node y ys :: yss) = if (x ≟ y) ∧ (n ≟ℕ length ys)
                                  then just (ys ++ yss)
                                  else nothing
```

Unlike the insert for the list patch, the insert for trees can fail. For it to succeed in inserting a node, there must be enough children that can be consumed from the list of trees. We check that by calling splitAt, a function from Agda's standard libraries that splits a list at a given position. The result is a pair consisting of ys and yss′. If there are insufficient elements in yss, then ys will contain less than n elements, and insert returns nothing.

The delete function ensures that the list is non-empty, as it did with the list version. Additionally, it checks that both the label and the arity of the root node at the head of the list match the label and arity to be deleted. If all checks pass, delete results in the tail of the list appended to the subtree list of the deleted node.

4.2 Diffing Trees

The diff algorithm on trees is defined as follows:

```
diff : List Tree → List Tree → Diff
diff []                 []                 = end
diff []                 (node y ys :: yss) =
  ins (y,length ys) (diff [] (ys ++ yss))
diff (node x xs :: xss) []                 =
  del (x,length xs) (diff (xs ++ xss) [])
diff (node x xs :: xss) (node y ys :: yss) =
  if (x ≟ y) ∧ (length xs ≟ℕ length ys) then best₃ else best₂
  where
  best₂ = del (x,length xs) (diff (xs ++ xss) (node y ys :: yss))
        ⊓ ins (y,length ys) (diff (node x xs :: xss) (ys ++ yss))
  best₃ = cpy (x,length xs) (diff (xs ++ xss) (ys ++ yss))
        ⊓ best₂
```

The structure of the function is very similar to the diff on lists (Section 3.2), only that now our inputs are lists of trees rather than lists of items. Focusing on the last case, we compare the labels and arities of the root nodes. We then select the best solution among the choices of deleting the source label, inserting the target label, or – when applicable – copying the label.

4.3 Discussion

There are several points to highlight regarding the tree diffing and patching implementations. One important difference between the list diff and the tree diff lies in the unit of work for an edit operation. Recall that the arguments to the former have the type List Item while those of the latter have the type List Tree. While the list diff operates on a single Item at a time, the tree diff does not operate on a whole Tree at a time. Instead, tree diffing works with nodes of the first tree. For example, the delete operation on the list node x xs :: xss removes the label x and prepends the xs to xss to get the new list of trees.

We also note an important similarity between the two diff implementations: both source and target are traversed in a fixed order. For lists, each element is considered in order, and for trees, we perform a depth-first preorder traversal. This similarity allows us to

reduce tree diff to a list diff by flattening the source and target trees in preorder and considering each item to be a pair of a label and an arity. This reduction makes it clear that we can employ the same dynamic programming approach that is known from list diff to the tree diff situation (see Section 6).

It is possible to use tree diff and patch with typed values. We transform the input into the untyped form and then parse the output. Unfortunately, this approach provides no guarantee that, if patching succeeds, we get a well-typed result. As an example, consider the following family consisting of two mutually recursive datatypes:

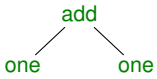
```
mutual
  data Expr : Set where
    add : Expr → Term → Expr
    one : Expr

  data Term : Set where
    neg : Expr → Term
```

Now, consider the following Diff with labels (reflecting the constructor names) and the appropriate arities:

```
badDiff = ins (add,2) $ ins (one,0) $ ins (one,0) $ end
```

Evaluating patch badDiff [] yields the singleton



which does not correspond to a well-typed expression. The tree patch and diff obey the patch-diff-spec; however, patch cannot exclude values such as badDiff even though they produce ill-typed terms. This is an issue that we aim to fix in the following section.

5. Families

Our goal now is to define implementations of patching and diffing that generically support families of (mutually recursive) datatypes with a representation that preserves types. We describe a universe to uniformly represent datatype families. Using this universe, we present a new Diff datatype and new, type-safe patch and diff functions. Later, we discuss how to implement the same functionality in Haskell for practical use (Section 7) as well as possible extensions to our approach (Section 8).

5.1 Universes

In order to write a generic diff algorithm that works on a large number of different datatypes, we first need a uniform description of a datatype. In the dependently typed world, this can be achieved using a universe construction [Benke et al., 2003, Morris, 2007, Oury and Swierstra, 2008]. A universe consists of a datatype of codes and an interpretation function that maps codes to types. We define generic functions by induction on the codes.

There are multiple universes suitable for generic programming. We use one that is the most appropriate for our purposes, one that corresponds closely to the labeled trees we considered in Section 4. Our codes describe a family of (mutually recursive) datatypes as an ordered, tree-like collection of lists. Contrary to other approaches, our view leads to a natural definition of Diff where ins, del, and cpy operate on constructors and lists. Our approach is a sum-of-products view, with sums and products of arbitrary arity. Compared to the oft-used binary sum-of-products view, we do not need to consider the restrictions of binary structure and nesting.

To begin defining the universe, we fix the number of datatypes in the family. Agda allows us to do so with a parameterized module:

```
module Codes (n : ℕ) where
```

Given n datatypes in a family, we can refer to a member with a number from 0 to n − 1. We define type synonyms for the index of a type in the family and for lists of these indices:

```
  TypeIx  : Set
  TypeIx  = Fin n

  TypeIxs : Set
  TypeIxs = List TypeIx
```

The type Fin n (defined in Agda's standard library) represents a finite range of n natural numbers with constructors zero and suc. For example, the possible values of type Fin 3 are zero, suc zero, and suc (suc zero).

5.2 Codes

The codes in our universe are described by the following types:

```
  Con  : Set
  Con  = TypeIxs

  Type : Set
  Type = List Con

  Fam  : Set
  Fam  = Vec Type n
```

A family Fam is described by a fixed-length vector of n types (one entry for each type member). A type Type consists of a list of constructors. Each field of a constructor is a reference to a type in the family, therefore Con is a list of type indices.

5.3 Representing a Family

Before we see how to interpret the codes as datatypes, let us first look at how to represent the family of expressions and terms from Section 4.3. While this is clearly a toy example, it nevertheless serves to illustrate two mutually recursive datatypes and constructors with various arities.

First, in order to encode the family, we declare a new module and open the Codes module, passing the number of datatypes forming our family.

```
module ExampleCodes where
  open Codes 2
```

For the sake of readability, we give meaningful names to the type indices:

```
  exprIx : TypeIx
  exprIx = zero

  termIx : TypeIx
  termIx = suc zero
```

Next, we give the codes for constructors and types. The type Term has just one constructor, neg (now encoded as 'neg'), and neg has an argument of type Expr (now encoded as exprIx).

```
  'neg' :  Con
  'neg' = exprIx :: []

  'term' : Type
  'term' = 'neg' :: []
```

We also define the codes 'add' and 'one' for the constructors add and one, respectively, and sequence them in 'expr' for the type Expr:

```
  'add' = exprIx :: termIx :: []
  'one' = []
  'expr' = 'add' :: 'one' :: []
```

Lastly, we group the descriptions 'expr' and 'term' together to build our 'example' family:

```
  'example' : Fam
  'example' = 'expr' :: 'term' :: []
```

Note that we must place the codes in the vector 'example' in the order indicated by the indices exprIx and termIx. This order is not enforced by the types.

5.4 Interpretation

We have specified codes, and now we define an interpretation function that maps the codes to types. As with Codes, we define an Interpretation module abstracting over the number of types in the family. We open the Codes module in order to use the codes while defining the interpretation.

```
module Interpretation (n : ℕ) where
  open Codes n
```

In order to define the interpretation function, we need a datatype of environments Env. Environments are heterogeneous lists parameterized by an interpretation function I and indexed by a list of codes:

```
data Env {A : Set} (I : A → Set) : List A → Set where
  []  : Env I []
  _::_ : ∀ {tx txs} → I tx → Env I txs → Env I (tx :: txs)
```

An environment of type Env I txs contains one element for each code in txs. By applying the interpretation function I to a code in txs, we get the type of the element in the environment. Note that Agda allows us to reuse the List constructors in Env: the context will serve to disambiguate between the two datatypes.

We use environments to store the fields of constructors. The fields have different types described by the constructor code Con. Here are the interpretation functions for Con, Type and Fam:

```
C⟦_⟧ : Con  → (TypeIx → Set)          → Set
C⟦_⟧ C I  = Env I C

T⟦_⟧ : Type → (TypeIx → Set)          → Set
T⟦_⟧ T I  = Σ (Fin (length T)) (λ c → C⟦ lookup c (fromList T) ⟧ I)

F⟦_⟧ : Fam → (TypeIx → Set) → TypeIx → Set
F⟦_⟧ F I t = T⟦ lookup t F ⟧ I
```

For each interpretation function, we assume that we have an existing interpretation function I. We will tie the knot shortly.

The function F⟦_⟧ accepts a family F and a type index t. In return, the function gives the interpretation of that type in this family. Recall that F is a vector of n types and that t ranges between 0 and n − 1. We can thus use lookup (from the standard library) to determine the Type for interpretation.

To get the interpretation of a type, we apply T⟦_⟧ to a type code T (a list of constructors). Semantically, the interpretation is the type of a constructor application (to zero or more arguments), which we represent with a dependent pair Σ. In a dependent pair, the second component is a function that takes the value of the first component. In our case, the first component of the pair is the type of a constructor index (constrained by the number of constructors in T), the second component is a function that returns the interpretation for the types of the arguments for that constructor, using C⟦_⟧. The function fromList turns a list into a vector.

Given a constructor code C, the function C⟦_⟧ returns the interpretation of its list of fields. Now, it is apparent why C⟦_⟧ uses the environment Env: a value of Env I C contains the interpretation of every field of C. It is for this reason that we need to pass the interpretation function I through each of level of the interpretation from the top.

We can now tie the knot using a fixed-point datatype for the interpretation function.

```
data μ (F : Fam) (t : TypeIx) : Set where
  ⟨_⟩ : F⟦ F ⟧ (μ F) t → μ F t
```

A value of type μ F t contains an isomorphic interpretation of a value of the type encoded by the index t in the family F. To give an intuition of how this works, we describe the value isomorphic to the constructor one.

```
one_μ : μ 'example' exprIx
one_μ = ⟨ suc zero,[] ⟩
```

The type of one_μ tells us that the family is given by 'example' and the type index is exprIx. The value inside the ⟨_⟩ is a dependent pair whose type is T⟦ 'expr' ⟧ (μ 'example'). The first component of the pair is the constructor index and the second component is the environment (which is empty since one has no fields).

5.5 Defining Diff

As we did with the descriptions for lists and trees, we look at the Diff datatype for families. First, we are generic in the family, so we abstract over a Fam using a parameterized module:

```
module GenericDiff (F : Fam) where
```

Note that we define the module GenericDiff nested within the Interpretation module. This means that we now abstract over both the number n and the family F of datatypes.

In the spirit of the tree diff, the generic diff operates on constructors of datatypes in the family F. The type of the constructor index depends on the type index:

```
ConIx : TypeIx → Set
ConIx t = Fin (length (lookup t F))
```

To find the valid range of constructor indices described by ConIx t, we look up the length of the list of constructors for the type index t.

We will find that we quite often need both a type index and a constructor index together; therefore, we define the following, convenient type synonym for this dependent pair:

```
Ixs : Set
Ixs = Σ TypeIx ConIx
```

We also introduce functions to access values from the Ixs:

```
typeix : Ixs → TypeIx
typeix (t,c) = t

fields : Ixs → TypeIxs
fields (t,c) = lookup c (fromList (lookup t F))
```

The function typeix projects out the type index, and fields determines the type indices of the constructor arguments (its "children").

We are now able to define the Diff datatype. It contains the familiar constructor names; however, the Diff datatype is now indexed over two lists of type indices, encoding the types of the source and target trees.

```
data Diff : TypeIxs → TypeIxs → Set where
  ins : {txs tys : TypeIxs} → (i : Ixs)      →
        Diff txs              (fields i ++ tys) →
        Diff txs              (typeix i :: tys)
  del : {txs tys : TypeIxs} → (i : Ixs)      →
        Diff (fields i ++ txs) tys          →
        Diff (typeix i :: txs) tys
  cpy : {txs tys : TypeIxs} → (i : Ixs)      →
        Diff (fields i ++ txs) (fields i ++ tys) →
        Diff (typeix i :: txs) (typeix i :: tys)
  end : Diff []              []
```

The extra type information allows us to read off how the constructors of Diff affect the types of the trees involved. For instance, the del constructor informs us that we can only delete a constructor (encoded by i) if the head of the source type indices list has the type typeix i. As with diff on untyped trees, deleting i will prepend its children (fields i) to the source list. Inserting a constructor has similar constraints on the target, and copying a constructor likewise affects both source and target.

To demonstrate the use of Diff, we revisit the badDiff example from Section 4.3. In order to translate it to the current setting, we define the constructor indices

```
oneIx : ConIx exprIx
oneIx = suc zero
```

```
addIx : ConIx exprIx
addIx = zero
```

and then consider the Diff

```
ins (exprIx,addIx) $ ins (exprIx,oneIx) $ ins (exprIx,oneIx) $ end
```

This Diff is now ill-typed. Looking at the partial Diff

```
ins (exprIx,oneIx) $ ins (exprIx,oneIx) $ end
```

we can see that it is of type

```
Diff [] (exprIx :: exprIx :: [])
```

i.e. a Diff that creates two expressions. On the other hand, the partial Diff

```
ins (exprIx,addIx)
```

has the type

```
∀ {txs tys} → Diff txs (exprIx :: termIx :: tys) → Diff txs (exprIx :: tys)
```

The types show that the latter expression expects a Diff producing an Expr and a Term and is therefore not compatible with the former.

It is interesting to note that we might have defined Diff as follows:

```
μEnv : TypeIxs → Set
μEnv = Env (μ F)

data Diff : ∀ {txs tys} → μEnv txs → μEnv tys → Set where
  ...
```

The μEnv function lifts the interpretation of the family μ F to an environment, such that this version of Diff is indeed indexed over actual values and not just their types. Which version to prefer is to some extent a question of taste. However, the version we presented reflects the amount of information we typically expect to be statically available: we expect to know the types of the trees involved in a change, but not the actual trees. Furthermore, we decided in Section 3 to consider a Diff datatype that completely determines both the source and the target tree. We describe variants of the Diff in Section 8 in which this is no longer the case. If Diff is indexed only by the types, such generalizations are easier. A final point is that the first variant allows the current version of Agda to infer type of a value of Diff. This is minor but useful in a practical sense.

5.6 Prerequisites for patch and diff

Before we can describe the implementations of patch and diff for families, we need a few utility functions for working with the types in our universe.

The first prerequisite is the ability for patch to manipulate environments in a few list-like ways. The need for this arises because we use μEnv (a synonym for Env (μ F)) to represent the source and target arguments in the patch and diff functions. Recall the uses of ++ and splitAt the tree versions of delete and insert, respectively. In a similar way, we need append and split operations for Env. For example, given environments Env I txs and Env I tys, we can construct the environment Env I (txs ++ tys) by appending the second environment to the first with the following function:

```
_+++_ : {A : Set} {I : A → Set} {txs : List A} {tys : List A} →
        Env I txs → Env I tys → Env I (txs ++ tys)
[]       +++ ys = ys
(x :: xs) +++ ys = x :: (xs +++ ys)
```

The definition is the same for list append, but it is typed for environments with the "shape" indicated in the signature.

The reverse operation is perhaps more interesting. Given an environment such as Env I (txs ++ tys) (whose shape is txs ++ tys), we want to split it into two environments of Env I txs and Env I tys. To achieve this, we define a function that splits the environment and passes the result to a continuation function.

```
splitEnv : {A R : Set} {I : A → Set} (txs : List A) {tys : List A} →
           Env I (txs ++ tys) → (Env I txs → Env I tys → R) → R
splitEnv []       xs       k = k [] xs
splitEnv (_ :: txs) (x :: xs) k = splitEnv txs xs (λ ys zs → k (x :: ys) zs)
```

The second issue we need to tackle is semi-decidable equality on indices. Consider the Item and Label types that we used for lists and trees. We described them as abstract, yet only requiring equality. This check was used when defining delete, and for the generic operation, we also need equality on constructor indices.

The standard library for Agda has a datatype for propositional equality. A value of type x ≡ y encodes the equality of x and y. Pattern matching on the only constructor refl informs the type checker that types x and y are equal.

We can define an equality function for Fin (the type of indices) that returns not just a Bool, but an optional equality proof:

```
_≟Fin_ : {n : ℕ} → (x y : Fin n) → Maybe (x ≡ y)
zero   ≟Fin zero   = just refl
(suc m) ≟Fin (suc n) with m ≟Fin n
...                  | nothing = nothing
(suc m) ≟Fin (suc .m) | just refl = just refl
_      ≟Fin _      = nothing
```

The dot pattern used for .m is Agda's notation indicating that the value m is determined by type constraints resulting from other parts of the pattern matching, in this case from the equality function.

The function _≟Fin_ will be useful for delete; however, we also need to check the index pair Ixs for diff. We can define a similar equality test:

```
_≟Ixs_ : (ix iy : Ixs) → Maybe (ix ≡ iy)
(tx,cx) ≟Ixs (ty, cy) with tx ≟Fin ty
...                   | nothing = nothing
(tx,cx) ≟Ixs (.tx,cy) | just refl with cx ≟Fin cy
...                                | nothing = nothing
(tx,cx) ≟Ixs (.tx,.cx) | just refl | just refl = just refl
```

We will see in the discussion of diff in Section 5.8 why a boolean equality is not enough.

We now have the equipment we need to write generic patch and diff.

5.7 Patching Families

The type of the patch function explicitly relates the additional information available in the Diff type to the types of the source and target trees:

```
patch : {txs tys : TypeIxs} →
        Diff txs tys → μEnv txs → Maybe (μEnv tys)
```

As we discussed above, we chose to index Diff only over the types of source and target trees, but not the actual values. This means that despite the additional type information, patch is still partial.

Once again, the definition of patch is the same as in Section 3.1. All we have to do is adapt the insert and delete functions. Let us look at insert first:

```
insert : {ts : TypeIxs} → (i : Ixs) →
         μEnv (fields i ++ ts) → Maybe (μEnv (typeix i :: ts))
```

The type dictates that the environment that is passed is of the shape fields i ++ ts. In order to get access to the fields of the inserted constructor, we need to split the environment into the fields i part and the part that is given by ts. This is where splitEnv comes in:

```
insert (t,c) xss = splitEnv (fields (t,c)) xss
                            (λ xs ys → just (⟨ c,xs ⟩ :: ys))
```

Note that like insert for lists but unlike that for trees, insert cannot

fail. The type signature dictates that suitable children are present, and so insertion always succeeds.

Next, consider delete:

```
delete : {ts : TypeIxs} → (i : Ixs) →
         μEnv (typeix i :: ts) → Maybe (μEnv (fields i ++ ts))
```

For the first time, our types are expressive enough to allow us to distinguish the signatures of insert and delete. Here, we know that our list of source trees is not empty. We can therefore match on the first element:

```
delete (t,c) (⟨ c′,xs ⟩ :: xss) with c ≟Fin c′
delete (t,c) (⟨ .c,xs ⟩ :: xss) | just refl = just (xs ++ xss)
...                             | nothing  = nothing
```

If the root constructor matches the one scheduled for deletion, we do so using our append function _++_ for environments. Otherwise, patching fails.

5.8 Diffing Families

Like patch, the diff function also operates on interpreted environments:

```
diff : ∀ {txs tys} → μEnv txs → μEnv tys → Diff txs tys
```

Note how the shapes of the environments determine the type of the resulting Diff.

The definition of diff is very similar to the tree version from Section 4.2:

```
diff {[]}     {[]}     []                []                =
  end
diff {tx :: _} {[]}     (⟨ cx,xs ⟩ :: xss) []                =
  del (tx,cx) (diff (xs ++ xss) [])
diff {[]}     {ty :: _} []                (⟨ cy,ys ⟩ :: yss) =
  ins (ty,cy) (diff [] (ys ++ yss))
diff {tx :: _} {ty :: _} (⟨ cx,xs ⟩ :: xss) (⟨ cy,ys ⟩ :: yss)
    with ((tx,cx) ≟Ixs (ty,cy))
... | nothing = ins (ty,cy) (diff (⟨ cx,xs ⟩ :: xss) (ys ++ yss))
                ⊓ del (tx,cx) (diff (xs ++ xss) (⟨ cy,ys ⟩ :: yss))
diff {tx :: _} {.tx :: _} (⟨ cx,xs ⟩ :: xss) (⟨ .cx,ys ⟩ :: yss)
    | just refl = ins (tx,cx) (diff (⟨ cx,xs ⟩ :: xss) (ys ++ yss))
                  ⊓ del (tx,cx) (diff (xs ++ xss) (⟨ cx,ys ⟩ :: yss))
                  ⊓ cpy (tx,cx) (diff (xs ++ xss) (ys ++ yss))
```

The main difference is that in the final cases, we make use of the equality test on index pairs that we defined in Section 5.6. It is this equality test that allows us to learn that tx and ty as well as cx and cy are equal. Knowing that they are the same tells us where to use the cpy constructor. Since we use the same type indices for the source and target in cpy, applying it in the fourth diff case would result in a type error.

6. Memoization

Up to this point, we have generalized the well-known list diff algorithm via an untyped tree diff algorithm to a generic typed diff that works on families of mutually recursive algebraic datatypes. However, all of the implementations we have presented are inefficient because we calculate the same subproblems many times. In all versions, the diff algorithm in the general case contains either two or even three recursive calls to itself, computing subproblems of the original problem and comparing them.

To make the algorithm more efficient, we need to restructure it in such a way that we can reuse solutions to subproblems as much as possible. The key insight that is being used for the list diff is that the order of subsequent insert and delete operations does not matter. Deleting a character and inserting one leads to the same subproblem as inserting first and then deleting. By arranging the subproblems in a two-dimensional table, it is possible to make this reuse precise.

The same idea is applicable to the tree diff and the generic diff. The reason is that the tree diff and hence the generic diff make use of a fixed traversal through the source and target trees, namely a depth-first preorder traversal. Thus, deleting and then inserting a label or constructor still leads to the same subproblem as inserting first and then deleting.

The tricky part that distinguishes the generic situation from both the list and tree diff settings is that if we try to place all the subproblems in a two-dimensional table, the table entries are of different types, as the recursive calls of diff are of different types. We therefore have to define a customized table datatype that makes it possible to keep track of the types of all the subproblems.

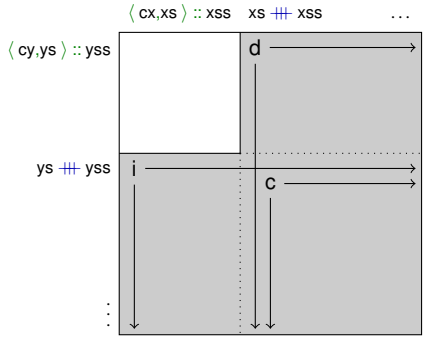
We should note at this point that despite the tabulation effort, the Agda version of our code will not get any more efficient. Currently, the Agda implementation makes use of a call-by-name evaluation strategy, and even when the Haskell backend is invoked, sharing can be lost. However, it is possible to express the tabulation idea in Agda, and just like the rest of the code, we have reimplemented this part in Haskell (see Section 7) where the sharing is respected and leads to a dramatic performance increase.

Laziness, on the other hand, is also helpful here. We can describe the tabulated algorithm in a straightforward way, and only those parts of the table will be computed that are actually required to determine the result of the main problem.

6.1 Table Datatype

When describing the table for the subproblems, we have to distinguish four different situations, depending on whether the source list of type indices is empty (nc), the target list is empty (cn), both lists are empty (nn), or both lists are non-empty (cc).

The cc situation is the most interesting one and is sketched in the following picture:



The source is of the form ⟨ cx,xs ⟩ :: xss of type μEnv (tx :: txs). The target is of the form ⟨ cy,ys ⟩ :: yss of type μEnv (ty :: tys). We can either delete cx, insert cy, or copy if cx and cy are the same constructor. These three operations lead to three different subproblems d, i and c, respectively, which involve the source list xs ++ xss and the target list ys ++ yss, each resulting from prepending the fields of a constructor to the list of remaining trees.

The picture shows that the c part can be shared when computing d and i – even if cx and cy are not the same constructor and copying is not possible.

Therefore, we define a table type DiffT where the cc always contains three subtables. When computing a value of this type, we will ensure that the c subtable is shared among the two others. If either the source or target list is empty, the situation is much simpler, because there is only one operation (either inserting or deleting) the diff algorithm can choose. Hence, only one subtable occurs in these constructors. And the nn case where both source and target lists are empty marks the lower right corner of the table.

There are no subtables in this case.

```
data DiffT : TypeIxs → TypeIxs → Set where
  cc : {txs tys : TypeIxs} (ix : Ixs) (iy : Ixs) →
       Diff  (typeix ix :: txs) (typeix iy :: tys) →
       DiffT (typeix ix :: txs) (fields  iy ++ tys) →
       DiffT (fields  ix ++ txs) (typeix iy :: tys) →
       DiffT (fields  ix ++ txs) (fields  iy ++ tys) →
       DiffT (typeix ix :: txs) (typeix iy :: tys)
  cn : {txs     : TypeIxs} (ix : Ixs)          →
       Diff  (typeix ix :: txs) []             →
       DiffT (fields  ix ++ txs) []            →
       DiffT (typeix ix :: txs) []
  nc : {    tys : TypeIxs}          (iy : Ixs) →
       Diff  []                (typeix iy :: tys) →
       DiffT []                (fields  iy ++ tys) →
       DiffT []                (typeix iy :: tys)
  nn : Diff  []                []              →
       DiffT []                []
```

All of the table entries also contain the actual Diff between the source and the target, and it is easy to define a function

```
getDiff : ∀ {txs tys} → DiffT txs tys → Diff txs tys
```

that extracts the Diff from a table.

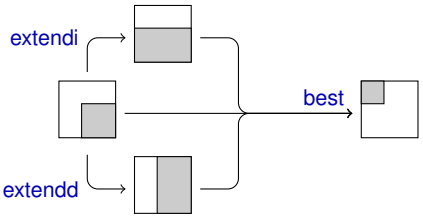
6.2 Building the Table

The next goal is to write a function

```
diffT : ∀ {txs tys} → μEnv txs → μEnv tys → DiffT txs tys
```

that builds a table for a given problem in a way that exploits sharing of subtables. We can then define a much more efficient diff function simply by calling diffT and applying getDiff to the resulting table.

Building the table is easy in an nn, nc or cn situation. For a cc situation, the strategy is as follows: we start by computing c (by calling diffT recursively), then use c in computing i and d, and finally select the best result from the subproblems in order to fill in the top-left corner of the table.



The picture illustrates the idea and the names of the functions performing the tasks. The code for diffT is shown below.

```
diffT {[]}     {[]}       []              []              =
  nn end
diffT {tx :: _} {[]}       (⟨ cx,xs ⟩ :: xss) []          =
  let d = diffT (xs +++ xss) []
  in cn (tx,cx)      (del (tx,cx) (getDiff d)) d
diffT {[]}     {ty :: _}  []              (⟨ cy,ys ⟩ :: yss) =
  let i = diffT [] (ys +++ yss)
  in nc (ty,cy)      (ins (ty,cy) (getDiff i)) i
diffT {tx :: _} {ty :: _}  (⟨ cx,xs ⟩ :: xss) (⟨ cy,ys ⟩ :: yss) =
  let c = diffT (xs +++ xss) (ys +++ yss)
      i = extendi c
      d = extendd c
  in cc (tx,cx) (ty,cy) (best i d c) i d c
```

Let us now look at the function best that as shown in the picture above makes use of the tree subtables to compute the top-left entry. It implements the algorithm we already know: The cost of performing insertion and deletion is compared and the best operation is selected; if the constructors are the same, copying is considered as well:

```
best : ∀ {txs tys tx ty} {cx : ConIx tx} {cy : ConIx ty} →
       DiffT (tx              :: txs) (fields (ty,cy) ++ tys) →
       DiffT (fields (tx,cx) ++ txs) (ty              :: tys) →
       DiffT (fields (tx,cx) ++ txs) (fields (ty,cy) ++ tys) →
       Diff  (tx              :: txs) (ty              :: tys)
best {_} {_} {tx} {ty } {cx} {cy } i d c
  with (tx,cx) ≟Ixs (ty,cy)
... | nothing = ins (ty,cy) (getDiff i )
                ⊓ del (tx,cx) (getDiff d)
best {_} {_} {tx} {.tx} {cx} {.cx} i d c
    | just refl = ins (tx,cx) (getDiff i )
                ⊓ del (tx,cx) (getDiff d)
                ⊓ cpy (tx,cx) (getDiff c)
```

The functions extendi and extendd take the shared part of the table and add another column in front or row on top, respectively. We only show extendi – the definition of extendd is analogous.

```
extendi : ∀ {txs tys tx} {cx : ConIx tx} →
          DiffT (fields (tx,cx) ++ txs) tys → DiffT (tx :: txs) tys
extendi {_} {[]}     {tx} {cx} d =
    cn (tx,cx) (del (tx,cx) (getDiff d)) d
extendi {_} {ty :: _} {tx} {cx} d = extracti d (λ cy c →
  let i = extendi c
  in cc (tx,cx) (ty,cy) (best i d c) i d c          )
```

When we want to add a column to the left, we have to know if there is more than one row in the table we get. Depending on that, we have to produce a cn or a cc. If the list of target trees is empty, i.e., if we are in the final row of the table, we can only continue by deleting. If there is more than one row in the table d we get, we can extract the table c with the first row removed. On c we can call extendi recursively to add a first column and get a table i. We now have tables i, d and c again that are suitable to build a cc entry, reusing the function best.

The last part missing is the function extracti. It takes a table consisting of multiple rows and drops the topmost row. The function makes the extracted part of the table available in a continuation argument. The implementation is simple: the type dictates that the table can only be an nc or cc. In both cases, the desired subtable is contained as a field of the constructor.

```
extracti : ∀ {R txs tys ty}                    →
           DiffT txs (ty :: tys)               →
           ((cy : ConIx ty) →
              DiffT txs (fields (ty,cy) ++ tys) → R) → R
extracti (nc (_,cy) _ i)       k = k cy i
extracti (cc _ (_,cy) _ i _ _) k = k cy i
```

The definition of our memoized generic diff is now complete.

7. Haskell

In this section we discuss a Haskell implementation of the generic diff algorithm. The Haskell version has the advantages of being far more efficient due to the use of actual sharing in the implementation. Furthermore, as a Haskell library it can actually be used in the context of larger applications. On the other hand, since the algorithm makes a significant use of dependent types, these concepts have to be encoded in Haskell using language extensions such as generalized algebraic datatypes (GADTs), type families and higher-ranked types. The resulting code is partially cluttered with traces of this encoding, which is why we have presented the Agda version in this paper. In this section, we do not show the complete Haskell implementation, but rather highlight the most important differences and show how we encoded the dependently typed aspects.

In particular, we have a look at how the universe is represented in Haskell, and how the Diff datatype looks like.

7.1 Representing Families

In Agda, we could just define codes that describe datatypes and then interpret them as types in a separate step. In Haskell, we have to use GADTs to define codes and their interpretation together. At the same time, we do somewhat more than in the Agda version. We provide a way to define a description for an already existing (and normally defined) family of mutually recursive datatypes such that we can use the generic algorithm with these standard datatypes. In contrast, the Agda interpretations of codes are only isomorphic, not identical to the datatypes defined by means of **data**. Establishing this isomorphism in Agda would be an additional step that we have not shown.

As an example family, let us once more use simple expressions:

```
data Expr = One
          | Add Expr Term
data Term = Neg Expr
```

Describing this family consists of multiple steps: first, we define a GADT that captures the structure of the family. Then, we make the GADT an instance of a class Family that provides several generic functions that we can then use in order to define the generic diff algorithm. As a final step, we associate the family GADT with the actual types contained in the family by means of a number of type class instances of class Type.

The GADT for the family looks as follows:

```
data ExampleFamily :: ∗ → ∗ → ∗ where
  'One' :: ExampleFamily Expr  Nil
  'Add' :: ExampleFamily Expr  (Cons Expr (Cons Term Nil))
  'Neg' :: ExampleFamily Term (Cons Expr Nil)
```

It plays the same role as 'example' in the Agda version. For each constructor in the family, we specify the result type and the list of fields by giving their types. Note that the fields are specified as a type-level list:

```
data Nil       = Nil
data Cons x xs = Cons x xs
```

The type ExampleFamily is then made an instance of the Family type class, which is shown below.

```
class Family f where
  decEq   :: f tx cxs → f ty cys → Maybe (tx:=:ty, cxs:=:cys)
  matcher :: f t cs → t → Maybe cs
  function :: f t cs → cs → t
  string   :: f t cs → String
```

The methods of class Family reflect the functionality we need in order to define generic diff. The function decEq corresponds closely to the equality on index pairs $\stackrel{?}{=}_{\mathsf{Ixs}}$ from Section 5.6. The differences are that this time, we do not compare indices, but actual types, and we return equality proofs in Haskell's equality GADT :=:. The function matcher can be used to test whether a value of type t belongs to a particular constructor, represented by f t cs. If matching succeeds, we get hold of the values of the fields of the constructor in the list cs. Conversely, function can be used to actually construct a t from arguments cs using a particular constructor. The last function string allows us to get a string representation of a particular constructor.

Defining the actual instance of Family for ExampleFamily is straightforward:

```
instance Family ExampleFamily where
  decEq 'One' 'One' = Just (Refl, Refl)
  decEq 'Add' 'Add' = Just (Refl, Refl)
  decEq 'Neg' 'Neg' = Just (Refl, Refl)
  decEq _     _     = Nothing

  matcher 'One' One       = Just Nil
  matcher 'Add' (Add e t) = Just (Cons e (Cons t Nil))
  matcher 'Neg' (Neg e)   = Just (Cons e Nil)
  matcher _     _         = Nothing

  function 'One' Nil                   = One
  function 'Add' (Cons e (Cons t Nil)) = Add e t
  function 'Neg' (Cons e Nil)          = Neg e

  string 'One' = "One"
  string 'Add' = "Add"
  string 'Neg' = "Neg"
```

As a final step, we have to instantiate the class Type:

```
class (Family f) ⇒ Type f t where
  constructors :: [Con f t]
```

This type class associates actual types with a family GADT. The method allows us to get the constructors specific to a particular datatype in the family. The datatype Con wraps the representation GADT such that the type of the fields of the constructor is hidden:

```
data Con :: (∗ → ∗ → ∗) → ∗ → ∗ → ∗ where
  Con :: (List f cs) ⇒ f t cs → Con f t
```

Wrapped by Con, the constructor representations for one datatype all have the same Haskell type and can therefore be put in a list. The class List ensures that cs is a type-level list where each element is a type of the family f again.

The instances we create for ExampleFamily are:

```
instance Type ExampleFamily Expr where
  constructors = [Con 'One', Con 'Add']

instance Type ExampleFamily Term where
  constructors = [Con 'Neg']
```

This is the complete boilerplate code we have to write in order to instantiate our Haskell generic diff library to an actual family of datatypes. While this is a significant amount, all of the code is straightforward to write and can even be automatically generated given the syntax tree of the datatype definitions, using a preprocessor or a meta-programming tool such as Template Haskell [Sheard and Jones, 2002].

7.2 Diff Datatype

The Diff datatype in Haskell is very similar to its Agda counterpart. Haskell lacks parameterized modules, hence – as with the generic functions – the datatype is explicitly parameterized over the family f.

```
data Diff :: (∗ → ∗ → ∗) → ∗ → ∗ → ∗ → ∗ where
  Ins :: (Type f t, List f cs,          List f tys) ⇒ f t cs →
         Diff f          txs  (Append cs tys)      →
         Diff f          txs  (Cons t    tys)
  Del :: (Type f t, List f cs, List f txs)       ⇒ f t cs →
         Diff f (Append cs txs)          tys     →
         Diff f (Cons t    txs)          tys
  Cpy :: (Type f t, List f cs, List f txs, List f tys) ⇒ f t cs →
         Diff f (Append cs txs) (Append cs tys)    →
         Diff f (Cons t    txs) (Cons t    tys)
  End :: Diff f Nil                Nil
```

The appending of type-level lists is described by the type-level function Append, which is defined with a type family:

```
type family    Append txs          tys :: ∗
type instance Append Nil          tys = tys
type instance Append (Cons tx txs) tys = Cons tx (Append txs tys)
```

7.3 Patching and Diffing

Using the Diff datatype above, it is quite easy to define patch in Haskell. For diff, however, some effort is required in order to

convince Haskell's type checker that the definition is okay. We make use of additional GADT witness arguments of type IsList that encode proofs that certain values are indeed type-level lists.

Furthermore, given a value of a Haskell datatype belonging to the family, we have to be able to discover the constructor of that value. We make use of a function

```
matchConstructor :: (Type f t) ⇒ t →
                    (∀cs.f t cs → IsList f cs → cs → r) → r
```

that takes a value of type t and a continuation that will be applied to the representation f t cs of the constructor that matches t.

Not only can diff be defined this way, but so can diffT and the extensions that we discuss in the next section.

We have instantiated the diff function for a datatype family that represents a JSON abstract syntax tree. The example shown in Section 1 is output from that function. The implementation also employs the extensions discussed in the following section.

8. Extensions

We have deliberately restricted ourselves to a simple setting in the exposition so far. There are several variations of the diff problem, more than we can possibly adequately discuss in the given space. However, we want to present two extensions to the simple algorithm above that we have implemented and that are rather essential in making the resulting library actually useful. Both extensions have been implemented in both the Agda and the Haskell versions.

Firstly, we show how the universe we use can be extended to cover constants, i.e., abstract types for which we have an equality function, but no further information about their structure.

Secondly, we discuss how patches can be compressed significantly by allowing the possibility to copy not only single constructors, but entire subtrees at once.

We mention more possible extensions when we discuss future work (Section 9).

8.1 Constants

While our universe can in theory represent types with very many or – using laziness – even infinite numbers of constructors such as Char and ℕ, it is clearly not efficient to use codes in this way. Instead, it is desirable to represent such types as abstract types in our family. To be usable in patching and diffing, an abstract type must admit an equality test. We therefore represent abstract types as a simple record, containing the type and the equality test for that type. In Haskell, we use a type class instead.

```
record Abstract : Set where
  field type    : Set
        decEq   : Decidable {type} _≡_
```

Usually, a Set field must not occur in a record of type Set, and we would have to give Abstract a larger type Set1. However, Agda has a flag to assume Set : Set, and we use it here in a non-essential way to save the work of having to lift most of our other definitions into Set1 as well.

We now have several choices for where to put the distinction between concrete and abstract types in our codes. Putting it in the Fam vector type (e.g. Vec (Either Type Abstract) n) works but is a bit cumbersome. We decided to convert Type, making it a datatype with two constructors.

```
data Type : Set where
  concr : List Con → Type
  abstr : Abstract → Type
```

In Haskell, we instead extend the Con type with a constructor Abs such that the instances of the Type type class can produce a singleton list containing an Abs constructor for abstract datatypes.

The behaviour for abstract types is as follows: each value of an abstract type is treated as a constructor with no fields. Because we have to compare two values in the diff algorithm in order to see whether we can copy, we need the equality function on the type. As a consequence of introducing abstract types, the following parts of the implementation need to be adapted:

- The interpretation function T⟦_⟧ becomes a datatype. We can then pattern match on the constructors of T⟦_⟧ in the implementation of diff and treat abstract types differently.
- The Ixs type becomes a datatype that in case of an abstract type saves the actual value of that type, since we do not have a code to represent it.
- The $\_\stackrel{?}{=}_{Ixs}\_$ function either uses the original $\_\stackrel{?}{=}_{Ixs}\_$ function or the equality test from the Abstract record.

8.2 Compression

When a complete (sub)expression can be copied, we can take a shortcut and not go through the trouble of traversing and copying each constructor. To support that, we extend the Diff datatype with a new constructor

```
data Diff : TypeIxs → TypeIxs → Set where
  ...
  cpyTree : ∀ {txs tys t} → Diff txs tys → Diff (t :: txs) (t :: tys)
```

In Section 3 we have discussed that our Diff datatype so far contains sufficient information to reproduce both the source and target trees. The point of using a Diff is usually to describe a change in an invertible, but relatively concise format. Large parts of the tree that remain unchanged are therefore rather uninteresting, hence we decide here to omit the copied tree in the cpyTree constructor. This is similar to the behaviour of the UNIX `diff` command, which also excludes information that is the same in both files in its output formats. If desired, an additional argument of type μ F t storing the value can easily be added.

Adapting patch to handle cpyTree occurrences is trivial.

We produce applications of cpyTree by post-processing values of type Diff generated by the diff algorithm, using a function

```
compress : ∀ {txs tys} → Diff txs tys → Diff txs tys
```

The compression function traverses the Diff recursively, replacing the cpy of a constructor with a cpyTree of its type whenever all the fields of the constructor are also copied using cpyTree.

Similar to cpyTree, we can also add insTree and delTree constructors that can insert or delete a whole tree at once. In the insTree case, we need to store the actual value being inserted. While this does not compress the size of the Diff as much as cpyTree does, it may still allow patch to work more efficiently.

9. Conclusions

This paper shows how diffing and patching evolve from the simple to the more general (and complex). Beginning with easily understood lists of items, we adapt the functions for labeled (untyped) rose trees. Then, utilizing a universe, we define type-safe diff and patch functions for families of (mutually recursive) datatypes. To improve the efficiency, we demonstrate the changes necessary for memoization. Our approach relies on features of advanced type systems in modern functional programming languages. The presentation of the code uses Agda for clarity and conciseness; however, we also discuss the implementation of a library in Haskell[4] . Lastly, we briefly mention some possible extensions.

There remains significant interesting work left to do. For example, we should experiment with cost functions. The previously

[4] We plan to release a Haskell package on the Hackage repository.

defined function simply counts the number of edit operations uniformly, but we might prefer edits that affect few larger parts over equivalent ones that affect many small parts. Alternatively, we might assign different costs to constructors of different datatypes.

There is a large amount of potential work with optimization, possibly at the cost of conceptual elegance. It is reasonable to employ heuristics for diff to find an edit script that is good enough, though not necessarily optimal.

Our generic diff implementation is based on algorithms for ordered trees. We should also consider unordered trees. This is helpful for expressing the fact that two operands of a binary operator have been swapped, which is more concise than deleting one of the operands and reinserting it at another place.

In general, it seems worthwhile to investigate the typical operations performed on typed tree data and consider ways to better describe such changes than on a per-constructor basis.

We will explore the use of our library more in real-word applications. This will help us experiment with various Diff types as well as test a larger collection of datatypes. One good use for the library is a structure-aware version control system. Another option is a graphical merge tool that visualizes the differences in structured documents and supports moving selected changes from one version into another.

Acknowledgments

We would like to thank Clara Löh, Thomas van Noort, members of the IFIP WG 2.1, and the anonymous reviewers for their invaluable comments.

References

Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for Generic Programs and Proofs in Dependent Type Theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

L. Bergroth, H. Hakonen, and T. Raita. A Survey of Longest Common Subsequence Algorithms. In *SPIRE 2000: Proceedings of the 7th International Symposium on String Processing and Information Retrieval*, pages 39–48, 2000.

Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful Change Detection in Structured Data. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, volume 26, pages 26–37, New York, NY, USA, June 1997. ACM Press.

Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change Detection in Hierarchically Structured Information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, volume 25, pages 493–504, New York, NY, USA, June 1996. ACM Press.

Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, July 2006.

Jeremy Gibbons. Datatype-Generic Programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, pages 1–71. Springer Berlin/Heidelberg, 2007.

Daniel S. Hirschberg. *The longest common subsequence problem*. PhD thesis, Princeton, NJ, USA, 1975.

Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic Views on Data Types. In Tarmo Uustalu, editor, *MPC 2006: Proceedings of the 8th International Conference on the Mathematics of Program Construction*, pages 209–234. July 2006.

J. W. Hunt and M. D. Mcilroy. An Algorithm for Differential File Comparison. Technical Report 41, Bell Laboratories Computing Science, July 1976.

Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 470–482, New York, NY, USA, 1997. ACM Press.

Philip N. Klein. Computing the Edit-Distance Between Unrooted Ordered Trees. In *ESA '98: Proceedings of the 6th Annual European Symposium on Algorithms*, pages 91–102. Springer-Verlag, London, UK, 1998.

Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.

Antoni Lozano and Gabriel Valiente. On the Maximum Common Embedded Subtree Problem for Ordered Trees. In *In C. Iliopoulos and T Lecroq, editors, String Algorithmics, chapter 7. King's College London Publications*, 2004.

Peter Morris. *Constructing Universes for Generic Programming*. PhD thesis, The University of Nottingham, November 2007.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, Göteborg, Sweden, 2007.

Ulf Norell. Dependently Typed Programming in Agda. In *Lecture notes of the 6th International Summer School on Advanced Functional Programming*, May 2008.

Bruno C. D. S. Oliveira, Ralf Hinze, and Andres Löh. Extensible and Modular Generics for the Masses. In Henrik Nilsson, editor, *Trends in Functional Programming*, volume 7 of *Trends in Functional Programming*, pages 199–216. Intellect, 2006.

Nicolas Oury and Wouter Swierstra. The Power of Pi. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 39–50, New York, NY, USA, 2008. ACM.

Luuk Peters. Change Detection in XML Trees: a Survey. In *3rd Twente Student Conference on IT*. Faculty of Electrical Engineering, Mathematics, and Computer Science, University of Twente, June 2005.

Dan Piponi. The Antidiagonal. http://blog.sigfpe.com/2007/09/ type-of-distinct-pairs.html, September 2007a.

Dan Piponi. Tries and their Derivatives. http://blog.sigfpe.com/2007/09/ tries-and-their-derivatives_08.html, September 2007b.

Alexey Rodriguez, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *Accepted to ICFP 2009*, 2009.

S. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, December 1977.

Tim Sheard and Simon P. Jones. Template Meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002.

Sjoerd Tieleman. Formalisation of version control with an emphasis on tree-structured data. Master's thesis, Universiteit Utrecht, August 2006.

Wuu Yang. Identifying Syntactic Differences Between Two Programs. *Software: Practice and Experience*, 21(7):739–755, 1991.

Kaizhong Zhang and Dennis Shasha. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.