Thesis for the degree of Master of Science

Generic type-safe `diff` and `patch` for families of datatypes

Eelco Lempsink

August 31, 2009

INF/SCR-08-89

Universiteit Utrecht

Center for Software Technology
Dept. of Information and Computing Sciences
Universiteit Utrecht
Utrecht, The Netherlands

Daily supervisor: Andres Löh

Second supervisor: Sean Leather

Contents

Chapter 1

Introduction

1.1 Motivation

The UNIX `diff` command finds the difference between two files and produces an edit script, describing the steps to get from the source file to the target file. The produced edit script can be used by the `patch` command to transform another similar source file to a file similar to the target file.

The ideas behind `diff` and `patch` are widely used, for example in version control systems [1, 2, 3, 4, 5]. A version control system keeps track of previous versions of files and is often used in a collaborative setting, where several people work on the same files. For the files that are managed, many current systems only make the distinction between plain text and binary content, even if the content they are managing has a structured interpretation, such as an XML file, a LaTeX document or source code.

In the domain of editors the merit of structured editors is often obvious. For example, when editing a document with the OpenOffice.org Writer, which uses XML as the underlying representation of documents, OpenOffice.org guarantees the XML structure of a document always stays intact.

Using plain text (e.g., line-based) edit scripts as produced by UNIX's `diff` to represent changes in files that have a structured representation is unsatisfying for three reasons: First, it is not always a clear representation to communicate what has changed to a user. For example, a small structural change might look like a many big changes when represented as a line-based edit script. Second, representing edit scripts – how to get from one version to another – for structured data as plain text is very fragile. If the script is slightly modified, applying it to a document might result in a file that does not contain a valid structure anymore. For example, the file does not parse as XML anymore. Third, even if the edit script does guarantee not to break the structure, the structure may still be invalid. In the case of XML, it may no longer adhere to its schema.

For XML files, there are several different algorithms and tools available that use of the structure of XML [23]. Even word processors, such as that offered by OpenOffice.org, have their own solutions for displaying and keeping track of differences between versions. However, for programming code and other structured documents – for which version control systems are often used – there is no general solution available.

1.2 Approach

We call the operation of calculating the edit script 'diffing' and applying an edit script 'patching'.

Our approach is to define a generic diffing and patching algorithm that works with type-safe edit scripts. Type-safe means that we make use of a type system to ensure the edit scripts are valid and do not break the structure when applied.

We use the dependently typed functional language Agda [20] for our code. Agda is well-suited for generic programming [22] and offers us a powerful type system. Its syntax is similar to Haskell. For the reader familiar with Haskell we offer Appendix A which has an overview of the major syntactic differences. To use our work on real-life we also 'ported' the algorithms to Haskell.

As an example, we look at the following two files and compare the output of UNIX's `diff` to the edit script from our solution.

```
if 2 < 0 then 0
         else 1
```

```
if 1 < 0
    then 2
    else 1
```

Using UNIX's `diff` command to find the difference between the two files, we get the following output:

```
@@ -1,2 +1,3 @@
-if 2 < 0 then 0
-         else 1
+if 1 < 0
+    then 2
+    else 1
```

If we look at the underlying structural difference between the two files we see that only the left side of the comparison in the `if` has changed and the result of the `then` branch. All other changes are merely formatting differences.

We parse the file into simple datatype representing the abstract syntax.

```
data Exp where
  If  :: Exp → Exp → Exp → Exp
  Val :: Num → Exp
  LT  :: Num → Num → Exp
```

Using this datatype (and the Num datatype, which is not shown), our diffing algorithm produces a (Haskell) value of the edit script that looks like this:

```
  Cpy 'If'
$ Cpy 'LT'
$ Ins 'l'
$ Del '2'
$ Cpy '0'
$ Cpy 'Val'
$ Cpy '1'
```

```
$ Cpy 'Val'
$ Ins  '2'
$ Del  '0'
$ End
```

While the edit script above may appear longer or more complex than the line-based edit script, it is actually more precise. Instead of deleting and adding lines with duplicated strings, we delete and copy syntax. It is important to note the amount of Cpy operations. Almost everything can be copied, only the actual numbers are replaced using Del and Ins.

1.3 Overview

To arrive at generic, type-safe diffing and patching for families of datatypes we first look at simple diffing and patching algorithms.

We start, in Chapter 2, with diffing and patching for lists. Our definitions are a simplification and generalization of UNIX's `diff` and `patch`. We look at how to define an edit script and a naïve algorithm. The definitions we show in Chapter 2 form the basis for further chapters.

Almost all datatypes can be represented as trees, the constructors forming the labels for the tree nodes and the arguments of the constructors defining how many subtrees a node has. Therefore, we define diffing and patching for labeled rose-trees in Chapter 3. We look at how the definitions have to change compared to the diffing and patching on lists.

When using the trees from Chapter 3 to represent datatypes, we find out we need more information on the types of the (constructor) nodes to define an edit script that satisfies the property we are after: an edit script that can cause an ill-typed value should be itself ill-typed.

In Chapter 4 we turn to datatype-generic programming, a technique commonly used to define generic functions, e.g., for pretty-printing, parsing, equality testing and ordered comparison. We define a *universe* to encode our types and use generic programming to define type-safe diffing and patching for datatypes.

The universe we defined in Chapter 4 we use in Chapter 5 to define a generic, type-safe implementation of diffing and patching. The diffing and patching algorithms do not have to change much to make the implementation type-safe. We change the definition of the edit script using dependently typed programming to guarantee that invalid edit scripts are ill-typed.

While chapter 5 is interesting for its theory, the algorithm is a too simplistic and inefficient to be useful in practice. The remaining chapters work towards making the solution more usable.

In Chapter 6 we define a memoized version of our algorithm, significantly improving on the exponential behaviour of the solution of Chapter 5. Creating a memoized version of the more simple, 'untyped' algorithms from Chapter 2 and 3 is straightforward. Because we have added dependent types to our edit script, the problem becomes much harder. We need to define our own typed memoization table in which to save subsolutions to subproblems.

The last step in making type-safe, generic diffing and patching usable in practice is redefining our solution in Haskell in Chapter 9. Using Haskell allows users to use a powerful, mature, general-purpose programming language and use existing libraries to represent the structure of files, e.g., JSON abstract syntax.

To make the algorithms usable in practice, we define an efficient version in Chapter 6, two extensions in Chapter 8 and Chapter 7, and a Haskell implementation in Chapter 9. We (need to) use quiet a few language extensions to do the same amount of dependently typed programming in Haskell as we used for our solution in Agda.

We end this thesis with a conclusion, in Chapter 10. We briefly review what we did and discuss related work and possible future work.

1.4 Contributions

The main contributions of this thesis are

- An implementation of generic type-safe diffing and patching algorithms for families of datatypes.
- A type-safe memoization technique.

Furthermore, this thesis contains several interesting use cases: generic programming in Agda, a list-view universe for types (Chapter 4), dependently typed generic programming in Haskel (Chapter 9).

Last, but not least, we plan to release our Haskell library on Hackage, the repository of Haskell library and program package, to the general public.

Chapter 2

Lists

UNIX's `diff` and `patch` work on files by treating them as a sequence of lines. In this chapter, to understand how diffing and patching work, we define implementations of diff and patch on lists of items in Agda. These implementations are a simple abstraction from UNIX's `diff` and `patch` implementation where an item is always a line of text. We also define a data structure to represent an edit script.

Finding an edit script is computationally equivalent to calculating the edit distance. The edit distance is the number of (primitive) operations in the edit script. For lists, calculating the edit script is equivalent to finding a solution for the longest common subsequence problem [7, 14].

Many of the definitions in this chapter are extended and reused in the following chapters.

2.1 Longest common subsequence

The longest common subsequence (LCS) problem can be succinctly described: given a set of sequences, find the longest combination of subsequences that all sequences have in common. As an example, with sequences of characters (strings), the LCS of `"aebcd"` and `"abedf"` is `"abd"`. The subsequences must occur in the same order, in this case "<u>a</u>e<u>b</u>c<u>d</u>" and "<u>a</u>b<u>e</u><u>d</u>f".

The problem of finding the LCS is solvable in polynomial time [7], with a straightforward algorithm. Better algorithms exist, reducing the computational complexity, but we present an unoptimized version.

```
lcs : List Item → List Item → List Item
lcs []         _          = []
lcs _          []         = []
lcs (x :: xs) (y :: ys)   = if x == y
                            then x :: lcs xs ys
                            else longest consume_x consume_y
  where consume_x = lcs      xs (y :: ys)
        consume_y = lcs (x :: xs)     ys

        longest : List Item → List Item → List Item
        longest xs ys = if length xs ⩽ length ys then ys else xs
```

In the case that one of the lists is empty, the common subsequence is an empty list. In the other case – both lists contain at least one item – we check whether the first items are the same. If the first item of both lists is indeed the same, we prepend it to the result of the recursive call the lcs function. If, however, the first items are different we try two different subsolutions: either the first item of the xs list is consumed or the y is dropped from the ys list. Both subsolutions are evaluated and the longest is used as the result.

As an example, assuming Item is a character, the above algorithm returns `"abd"` when called as lcs `"aabcd"` `"abdef"`. The longest common subsequence is not necessarily unique. For instance lcs `"abcdbdb"` `"cbacbaba"` could give both `"bcbb"` and `"acbb"` back as a result. The code above gives the first answer, because of how longest is defined and called. We are not interested in finding *all* longest common subsequences: because the goal is to create a minimal patch, so any maximal solution will do.

2.2 Edit script

To turn the lcs algorithm into the diff algorithm we modify it to calculate an edit script instead of the longest common subsequence. An edit script contains a sequence of edit operations. The result of the diff function is such an edit script, describing the operations to get from the source list to the target list.

We define the Diff datatype to represent an edit script.

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

The first three constructors of the Diff datatype represent the different possible operations: inserting (ins) an item in the list, deleting (del) an item from the list or copying (cpy) an item. The last constructor (end) is the base case for the recursive definition, indicating the end of the edit script.

We construct a value of type Diff by recursive application of the constructors. An alternative approach is to define a datatype for the operations and make Diff a List of those operations. In the next chapters we extend the edit script and add more precise types; the type of the recursive Diff argument will depend on the operation. If we use a simple List, capturing this dependency is not possible.

An edit script describing the changes between the strings `"moo"` and `"cow"` (Item is a Char in this case) can be written as

```
del 'm' $
ins 'c' $
cpy 'o' $
del 'o' $
ins 'w' $
end
```

Taking the string `"moo"` and patching it with the above edit script yields the string `"cow"`. The edit is readable for a human and patching the string `"moo"` by hand is a simple task.

The choice for these operations is not arbitrary. Inserting and deleting are necessary operations for the edit script to be usable. By adding the cpy operation we make the problem of finding the shortest edit script non-trivial. Otherwise all edit scripts can simply delete the complete source and then insert the complete target. More advanced types of edit scripts are not used in the thesis, although some are discussed in Section 10.1.

The arguments for each constructor of the edit script contain enough information to be able to reconstruct both the target and source list from the edit script. A variation is to leave out the Item information in the del and cpy information. Leaving out Items makes the patch function more forgiving of its input (if the source list does not exactly match the original source, it might still succeed), but it also means we can no longer invert the edit script. The inverted edit script is a description of how to get from the target to the source. Inverting an edit script with our definition is a simple function replacing each ins with a del and the other way around. Invertible edit scripts are useful in the context of distributed version control systems, such as Darcs [2].

Using the edit script, we define the diff and patch functions.

2.3 Diffing

The structure of the naïve diff algorithm is a simple modification of the lcs algorithm.

```
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
```

The behaviour for empty lists is slightly different. While the lcs algorithm returned the empty list when either the source or target list was empty (or both), diff only ends after both the source and target list are completely consumed.

Next, consider the case where both lists have at least one item, but the items are different: the result is best_2. Comparing the diff algorithm with lcs, note that the del subsolution is the same as consume_x, and the ins subsolution the same as consume_y. In other words, inserting an item is equivalent to consuming an item from the target, while consuming an item from the source is equivalent to delete operation in the edit script.

The ⊓ operator is similar to longest from the lcs algorithm.

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

By using ⊓ we choose the edit script that has the minimal cost. The definition of our cost function determines what the minimal cost means. To find the shortest possible edit script, we simply define each operation to have cost 1.

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

The goal of finding the shortest edit script is equivalent to the goal of the lcs algorithm, which finds the maximal number of equal parts of a sequence.

Finally, the last case of the diff algorithm is that both lists have at least one item and the first item is the same for both lists. In this case, the result is best_3, choosing the best subsolution out of either deleting, inserting or copying the item. Abstracting out the (implicit) cost function makes the diff algorithm more flexible than the lcs algorithm defined previously. By defining diff with best_3 we can use different cost functions to get different edit scripts. The lcs algorithm always copies when two items are the same.

To see how an edit script is used, we look at patch.

2.4 Patching

The patch function takes a Diff and a value and *may* produce a patched value.

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

If the Diff contains an operation that cannot be fulfilled, e.g., the item of del operation does not match the item in the source list, patching fails. In combination with diff, however, the following property should hold:

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

When patch is applied to the result of diff and the same source list (xs) as diff, it returns a list identical to the target list (ys) argument of diff.

The patch function is straightforward to implement. We use a slightly more general definition than is needed at this point, which allows us to unify the definition of patch across chapters. Only the insert and delete functions differ in the following chapters.

```
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
```

The operator _⋄_ is monadic composition on Maybe. If the patch fails anywhere, nothing is propagated as the result.[1]

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

[1]Refer to Appendix A for Agda syntax explanations

Let us look at each of the cases for patch and the implementations of insert and delete.

$$
\begin{aligned}
&\mathsf{insert} : \mathsf{Item} \to \mathsf{List\ Item} \to \mathsf{Maybe}\ (\mathsf{List\ Item}) \\
&\mathsf{insert}\ \mathsf{x}\ \mathsf{ys} = \mathsf{just}\ (\mathsf{x} :: \mathsf{ys})
\end{aligned}
$$

In the ins case, we add the item as the head of the list. Later versions of insert can fail, so we already use Maybe here to be able to keep the type signatures similar and the definition of patch the same.

In the del case, if the input list is empty or the expected item is not found in the target, delete returns nothing.

$$
\begin{aligned}
&\mathsf{delete} : \mathsf{Item} \to \mathsf{List\ Item} \to \mathsf{Maybe}\ (\mathsf{List\ Item}) \\
&\mathsf{delete}\ \mathsf{x}\ [\,] \quad = \mathsf{nothing} \\
&\mathsf{delete}\ \mathsf{x}\ (\mathsf{y} :: \mathsf{ys}) = \mathsf{if}\ \mathsf{x} == \mathsf{y}\ \mathsf{then}\ \mathsf{just}\ \mathsf{ys}\ \mathsf{else}\ \mathsf{nothing}
\end{aligned}
$$

Otherwise, the item is discarded and the tail of the target list returned.

The case for cpy uses both insert and delete. Deleting is done before the applying the rest of the edit script, therefore operating on the source list. Inserting is done after patching, when the source list has been transformed into a (partial) target list.

The end case only succeeds if both the source and target list have been processed completely.

2.5 Discussion

The implementation of the diff algorithm as presented above is very inefficient. Because we have multiple recursive calls at every step, without sharing of subresults, the complexity of the algorithm is exponential. There are several ways to improve this naïve algorithm.

- Given the cost and $\_\sqcap\_$ functions above, we can simply replace best_3 by the cpy operation 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. Compare this behaviour to the lcs function, which always includes an item if it can be copied.

- Using a dynamic programming or memoization approach, we can share recursive calls as much as possible. In Chapter 6 we show how to use memoization for an evolved version of the algorithm presented in this chapter.

- 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. To be able to use lazy comparison we need to implement the algorithm in a language supporting lazy evaluation, such as Haskell. In Chapter 9 we do show a Haskell implementation that benefits from lazyness.

For now, we defer the efficiency issues until Chapter 6 and focus on clarity of representation for the next chapters.

In Chapter 3 we look how to extend the diffing and patching algorithms to work with trees and in Chapter 5 we show how to extend the algorithms furither to work with type-safe edit scripts.

Chapter 3

Trees

Trees are generic way to represent (almost) all datatypes. Values of datatypes can be represented as a tree by using the constructors as labeled nodes and the arguments of the constructor (which we also call *fields*) as the subtrees. Our goal is to define diffing and patching for datatypes, by using trees.

To do diffing and patching on trees, we need an algorithm similar to the longest common subsequence algorithm from the last chapter. As in the previous chapter, finding the algorithm is not part of our research, but we implement the algorithm in a more general way than previously published.

The research area for complex structured data is much wider than that of sequential data. There are several variations of the problem of calculating the tree edit distance. For instance, the edit operations may include inserting, deleting, updating, or copying single nodes or entire subtrees. The trees may be ordered or unordered, labeled or unlabeled, rooted or unrooted [28, 33, 8]. Furthermore, the same problem is sometimes researched in different fields and may even have different names.

The research fields that produced the most relevant work for our problem are (meaningful) change detection [10, 9], XML [23] diffing and program syntax [32]. Also, in the context of syntax-directed version control [30] an algorithm to do diffing and patching is needed.

Considering the amount of research done previously, it almost seems to suggest we should be able to translate our problem to an existing solution, such as a simple XML format. However, as this chapter and the next chapter show, to have a solution for diffing and patching datatypes in a type-safe way, we need to have a powerful type system. We therefore first implement a simple algorithm for trees in this chapter. This implementation provides a clue of what we need to do to make the algorithm suitable for datatypes: add more descriptive types.

The trees we use are labeled ordered rose trees. Each node in a rose tree has an arbitrarily-sized forest of subtrees.

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

The implementation of Label is not important. As for Item in the previous chapter, we only require an implementation of an equality test for Label. The structure of the rose tree is suitable to represent regular datatypes, the Label being the name of the constructor and the subtrees the fields of constructor. In Section 3.5

Figure 3.1: Contracting edges

we show an example how such a tree can be used as an (untyped) datatype representation.

3.1 Maximum Common Embedded Subtree

We can use the same approach as we did in the previous chapter. By modifying an algorithm that finds the largest common tree (similar to the longest common subsequence) we can construct an algorithm that calculates an edit script.

The problem for finding the largest common tree is called the Maximum Common Embedded Subtree (MCES) problem by Lozano and Valiente [17]. Given a set of trees, we need to find the largest possible tree contained in all of them. As with finding the subsequence, the tree does not have to be one part in both source and target, but may be constructed from several subtrees.

The MCES algorithm presented by Lozano and Valiente [17] is based on work by Klein [16] and works with ordered, untyped trees. The nodes do not have a value, so there is no value to have a type. Our presentation is adapted to include a label for each node, of a single type, a trivial extension.

The key idea of the MCES algorithm is the contraction of edges that represents the insert and delete operations. As with lists, consuming 'something' from the source is deleting, consuming 'something' from the target is the insert operation. For the MCES algorithm, that 'something' is an edge, and consuming an edge means contracting it. Contracting is demonstrated in Figure 3.1. The dashed edge is contracted, one of the nodes is deleted and the result is the right tree.

3.2 Edit script

In the MCES algorithm as presented by Lozano and Valiente the trees are serialized to a sequence before the operations are applied. We believe the serialized representation obscures the general algorithm and we can do better. We only keep the depth-first preorder traversal of the serialization and choose suitable data structures.

3.2.1 Datatype

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

```
data Diff : Set where
  ins  : Label × ℕ → Diff → Diff
```

$$\begin{aligned}
\mathsf{del} &: \mathsf{Label} \times \mathbb{N} \to \mathsf{Diff} \to \mathsf{Diff}\\
\mathsf{cpy} &: \mathsf{Label} \times \mathbb{N} \to \mathsf{Diff} \to \mathsf{Diff}\\
\mathsf{end} &: \mathsf{Diff}
\end{aligned}$$

Instead of an item, we pair the Label with a number representing the arity, the number of (direct) children for each node. We diverge from the original MCES algorithm, which is untyped, because we restrict the operations such that nodes not change arity.

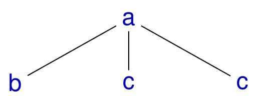
3.2.2 Stack

The diff and patch functions do not work directly with Tree arguments. Instead, they take a list of Trees, representing a stack-based approach to traversing the tree. The reason using stacks is that when we consume an edge and a node, we need to deal with multiple subtrees.

Consider the following example tree

$$\mathsf{node}\ \mathsf{a}\ (\mathsf{node}\ \mathsf{b}\ []\ ::\ \mathsf{node}\ \mathsf{c}\ []\ ::\ \mathsf{node}\ \mathsf{c}\ []\ ::\ [])$$

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

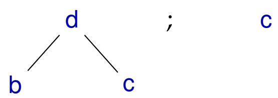


Applying the partial diff $\mathsf{del}\ (\mathsf{a}, 3)$ to the tree above (which removes the node labeled a) results in multiple trees, the three children of the node a:

b ; c ; c

In terms of stack operations: we pop a tree with root a and 3 children, and we push each of those children back to the stack.

When we now want to add another label to the tree, we have to figure out how many of the trees on the stack become children of that node. Here is where we need the arity. For example, applying the partial diff $\mathsf{ins}\ (\mathsf{d}, 2)$ pops two trees from the stack and pushes one back, resulting in two trees:



One might argue that the arity is not needed for the del case. This is true, but we include it for the same reason we include the label of the node in the del case, which is also not strictly needed: to check if the node to delete matches with the node we expect.

3.2.3 Example

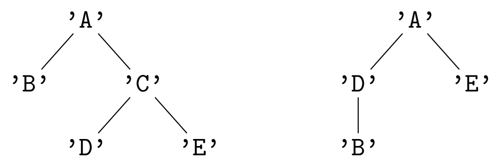
We look at an edit script describing the changes between two trees. As Labels we use characters. The source tree contains five nodes, labeled from `'A'` to `'E'`:

```
sourceTree = node 'A' (node 'B' [] ::
                       node 'C' (node 'D' [] ::
                                 node 'E' [] :: []) :: [])
```

The target tree contains 4 of the 5 nodes. Node 'C' has been deleted and the tree is built differently.

```
targetTree = node 'A' (node 'D' (node 'B' [] :: []) ::
                       node 'E' [] :: [])
```

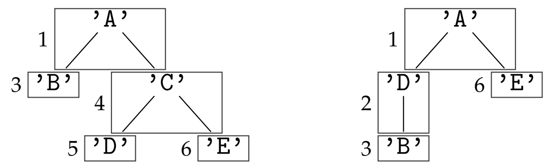
We can graphicly display both trees:



The result of the diff function, defined below, on those trees is the following edit script:

```
1. cpy ('A' , 2) $
2. ins ('D' , 1) $
3. cpy ('B' , 0) $
4. del ('C' , 2) $
5. del ('D' , 0) $
6. cpy ('E' , 0) $
7. end
```

The code above is a single edit script, with line numbers used to number each operation. We depict each step in the graphic below.



The first operation, cpy ('A' , 2), consumes the 'A' node from both the source and the target. The arity, 2, indicates that the result of this cpy operation is two subtrees. The second operation, ins ('D' , 1), consumes the node 'D' from the target, leaving a single subtree. The del operation (step 4 and 5) consume nodes from the target tree. Note how the node 'D' is both inserted and deleted, but with a different arity. We can only copy a node if the arity does not change, e.g. the node 'B' in this example.

In the next section we define how the diff algorithm for trees works.

3.3 Diffing

The structure of the diff algorithm is similar to the version for lists, shown in Section 2.3. Note, however, that diff works with stacks of trees, in this case for both the source and the target tree.

```
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
```

We calculate the arity simply with the length function. In the last, most interesting case we check both the label and the arity for equality when deciding whether to use cpy or not.

Note an important similarity between this diff implementation and the one for lists: both source and target are traversed in a fixed order. We start at the root of the first tree on both stacks and recursively visit all the child nodes, doing a depth-first preorder traversal, thereby reducing the tree diff to a list diff. The reduction is similar to the serialization used by Lozano and Valiente and it also indicates that we can use a standard dynamic programming approach to get to an efficient implementation of the diffing algorithm.

3.4 Patching

The type signature of the patch function shows that patch also works on Lists of trees, representing the stacks.

```
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
```

Both the insert and delete function can fail in this case, unlike the implementation for lists. We use the function splitAt, a function from Agda's standard libraries that splits a list at a given position, to pop the right amount of subtrees from the stack. Because the function also works when n is bigger than the length of the lists, we need to check if it is within the bounds the of the list. The $==_{\mathbb{N}}$

function is an equality test for natural numbers. The insert fails if the arity of the label inserted is larger than the available subtrees. If insert succeeds, it uses the subtrees from the stack and pushes a new node on top of the stack.

For delete we inspect the topmost tree. We cannot delete from an empty tree, so we return nothing in that case. However, if both x == y and the arity of the root node of the topmost tree matches the expected arity, we can safely delete the node and push the subtrees onto the stack.

3.5 Discussion

The tree diff and patch defined in this chapter can be used for diffing and patching datatypes. We use the constructor names as labels. However, since the tree nodes do not contain any type information, we quickly run into problems.

Consider a family of two mutually recursive datatypes:

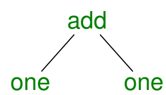
```
mutual
  data Expr : Set where
    add : Expr → Term → Expr
    one : Expr
  data Term : Set where
    neg : Expr → Term
```

This is an example family for the purpose of demonstration only, containing a small number of constructors, each of different arity. We encode the constructors as the labels in the tree, together with the arity. For example, $(\mathsf{add}_l, 2)$ is the encoding off the add constructor. We can not, however, encode the types of the constructors.

Now, consider the following Diff

$$\mathsf{badDiff} = \mathsf{ins}\ (\mathsf{add}_l, 2)\ \$\ \mathsf{ins}\ (\mathsf{one}_l, 0)\ \$\ \mathsf{ins}\ (\mathsf{one}_l, 0)\ \$\ \mathsf{end}$$

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, but patch cannot exclude values, such as badDiff, that produce ill-typed terms. In the Chapter 5, we revisit this example family and show how we can restrict diff and patch by constructing a more refined Diff type.

Chapter 4

Universe

This chapter is an intermezzo connecting the tree diffing and patching to type-safe implementations. In the previous chapter, we demonstrated how using tree diffing and a simple encoding of constructors was not sufficient to achieve generic, type-safe diffing and patching. In the next chapter, we will use generic programming to give a better, generic, type-safe implementation of diffing and patching for datatypes. To get there, we define an encoding of constructors and their types, suitable for datatype-generic programming [13], in this chapter.

Almost all generic programming libraries use a *generic view* to represent the structure of datatypes [15]. The most popular is the sum of products view – e.g. in Extensible and Modular Generics for the Masses [21] – which allows generic functions to be defined by induction on the structure.

The generic view is one part of a *universe* [6, 19, 22]. A universe consists of a datatype of *codes* (the view) and an interpretation function that maps the codes to types.

4.1 Encoding

There are multiple universes suitable for generic programming. We use one that corresponds closely to the labeled trees we considered in Chapter 3.

Consider again the datatypes we used as an example in Section 3.5.

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

A group of datatypes is also called a *system* or a *family*. The family formed by the datatypes above contains two mutually recursive datatypes with constructors of different arities. We use this family as an example to explain the encoding we chose.

The encoding is based on simply numbering the types and constructors. Each unique type has a fixed index in a family. The indices are of type TypeIx, which we explain later. For the example, we give the indices readable names.

```
exprIx : TypeIx
exprIx = zero

termIx : TypeIx
termIx = suc zero
```

A type index, however, is not yet a type encoding. We use the indices later to *refer* to types in the encoding of the family.

To encode a constructor we use a list for the arguments (or *fields*) of the constructor. Each argument is encoded as a type index, as defined above. We encode the constructor neg (of the type Term) as 'neg'. The argument of neg is of type Expr, which we refer to with exprIx.

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

Similarly, a type has a number of constructors. To encode a type, we define an encoding for each constructor and group them together in a list to encode the type. The type Term has only one constructor.

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

The type Expr is defined similarly:

```
'add' : Con
'add' = exprIx :: termIx :: []

'one' : Con
'one' = []

'expr' : Type
'expr' = 'add' :: 'one' :: []
```

We use the encoding of the types to encode the family. To guarantee that the type indices are unique and to be able to look up a type encoding using a type index, we group all the types in the (example) family together in a value of type Fam.

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

This concludes the example. For the types of the encoding of the example, Con, Type, Fam, etc., we create a module Codes.

```
module Codes (n : ℕ) where
```

The module is parameterized by the natural number n, abstracting over the size of the family. The parameterization makes n available in the scope of the module and we use it for the Fam type, which is a vector of Types:

```
Fam : Set
Fam = Vec Type n
```

The type of Type is simply a List of Con and the Con is a List of TypeIx for which we use the alias TypeIxs.

```
Type : Set
Type = List Con
Con : Set
Con = TypeIxs
```

Now, for the TypeIx we use the n again to construct the type Fin n[1]. By reusing n we restrict the type indices to the number of datatypes in the family and thereby guarantee that a type index always points to a type available in the family.

```
TypeIx  : Set
TypeIx  = Fin n
TypeIxs : Set
TypeIxs = List TypeIx
```

Note that the placement of the codes for types in a Fam vector must match the indices of those types; this is not enforced by the type system.

4.2 Interpretation

The second part of a universe is the interpretation function. The interpretation function converts codes to types.

We used three encodings: for constructors (Con), types (Type) and the family (Fam). For each of those types we need an interpretation function.

Not only are we interested in calculating types from codes, but we also need a way to write values of those types. We can use an *environment* to store values with types calculated from codes in a heterogeneous list-like structure.

4.2.1 Environments

We introduce a small example universe to demonstrate how an environment works. We use a simple datatype, Code, for encoding natural numbers and booleans.

```
data Code : Set where
  N : Code
  B : Code
```

An interpretation function maps the codes to types. We define a straightforward function and use the actual types for natural numbers and booleans.

```
interpretation : Code → Set
interpretation N = ℕ
interpretation B = Bool
```

Note that interpretation is a type function, its result is a type (of type Set).

Using an *environment* we can use Code and the interpretation function to create a heterogeneous list with items that are either a natural number or a boolean.

[1] An explanation of the Fin type can be found in Appendix A.

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)
```

We use the same constructors as List, but the type of an element in the environment is the result of applying the interpretation function I to a code tx.

For our example, the type of the codes (A) is Code and I therefore has the type Code → Set, the type of the interpretation function defined above. We define environment as an example of a heterogeneous list using a list of Codes and our interpretation function.

```
environment : Env interpretation (N :: B :: B :: N :: [])
environment = 4 :: false :: true :: 2 :: []
```

Next, we look at interpreting the codes we defined to encode families of datatypes.

4.2.2 Interpretation of families

In this subsection we look at a more complex example of interpretation, using again the example datatype that we used at the beginning of this chapter.

The goal of the interpretation is to create types and values isomorphic to the types and values we encoded. We do not explicitly establish the isomorphism between the encoded family and the interpretation of the codes, but it is straightforward to see that the isomorphism holds.

We use a dataype μ to construct values using the encoding of a family and a TypeIx. The function typeInterp is used to calculate the type of the argument to the ⟨_⟩ constructor. For now, we can think of μ being a simple container to hold interpreted values.

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

We define the implementation of typeInterp later, after we have constructed all the necessary building blocks.

Using μ, we write the interpreted types of the 'example' family.

```
Expr_μ : Set
Expr_μ = μ 'example' exprIx
Term_μ : Set
Term_μ = μ 'example' termIx
```

To write a function that is isomorphic to an encoded constructor, we use the ⟨_⟩ constructor of the μ datatype. For example the add_μ constructor:

```
add_μ     : Expr_μ → Term_μ → Expr_μ
add_μ e t = ⟨ ... ⟩
```

The '. . .' is not an Agda language construct, but indicates a blank spot. What has to be filled in at the '. . .' depends on the implementation of typeInterp, but we can already imagine that we need two things that are unique for each constructor: the index of the (encoded) constructor and the arguments. We get the arguments passed as e and t, but we also need a way to store them.

For the constructor indices, we define a helper function to calculate the type of the constructor indices given a type index.

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

We use lookup to retrieve the encoding of a type from the encoding of the family. The lookup function is defined in Agda's standard library as

```
lookup : ∀ {A n} → Fin n → Vec A n → A
lookup zero    (x :: xs) = x
lookup (suc i) (x :: xs) = lookup i xs
```

Note how the n is shared by the Fin and the Vec type. Because the value of the Fin can not be bigger than the size of the Vec, the lookup always succeeds. The result of the lookup in our case is the encoding of a type: a list of its constructor encodings. By using the length of that list we restrict the value of the constructor indices, ensuring they can be safely used to do lookups in the list of constructors.

Using ConIx, we define readable names for the constructors indices:

```
addIx : ConIx 'example' exprIx
addIx = zero

oneIx : ConIx 'example' exprIx
oneIx = suc zero

negIx : ConIx 'example' termIx
negIx = zero
```

We can use a constructor index to retrieve the constructor encoding from the family if we also have the type index:

```
lookupCon : (F : Fam) → (t : TypeIx) → ConIx F t → Con
lookupCon F t c = lookup c (fromList (lookup t F))
```

Note that we have to convert the list of constructor encodings to a vector using fromList to be able to use lookup. The fromList function is also defined in Agda's standard libary:

```
fromList : ∀ {A} → (xs : List A) → Vec A (length xs)
fromList List.[]          = []
fromList (List._::_ x xs) = x :: fromList xs
```

We return to filling in the '. . .' in the definition of add$_\mu$. To store the arguments and match the encoding of the constructor we can use the Env datatype we defined in the previous section. The codes for the Env are TypeIxs, which means the interpretation function must have the type TypeIx → Set. We construct such an interpretation function by applying μ to 'example'.

Using the arguments passed to the add_μ function is passed, we can write the interpretation of the arguments with the following type:

```
addArgs : Exprμ → Termμ →
          Env (μ 'example') (lookupCon 'example' exprIx addIx)
addArgs e t = e :: t :: []
```

We group the index of the constructor and the interpretation of the arguments in a dependent pair, using the Σ datatype from Agda's standard library.

```
data Σ (A : Set) (B : A → Set) : Set where
  _,_ : (x : A) (y : B x) → Σ A B
```

The type of the second argument of the _,_ constructor depends on the value of the first argument. With the dependent pair, the '...' of the add_μ definition can be written as:

```
addPair : Exprμ → Termμ →
          Σ (ConIx 'example' exprIx)
            (\c → Env (μ 'example') (lookupCon 'example' exprIx c))
addPair e t = addIx , e :: t :: []
```

Note that we constructed a type that is usable for all constructors encodings of Expr. Compare the type above with the one for onePair.

```
onePair : Σ (ConIx 'example' exprIx)
            (\c → Env (μ 'example') (lookupCon 'example' exprIx c))
onePair = oneIx , []
```

There is quite some repetition in the type signature of the addPair and onePair functions. We can easily abstract out both the family ('example') and the type index (exprIx) using the typeInterp function.

```
typeInterp : Fam → TypeIx → Set
typeInterp F t = Σ (ConIx F t) (\c → Env (μ F) (lookupCon F t c))
```

This completes the definition of the interpretation functions. We can now fill in the '...' in the definition of add_μ concisely, without using the addPair or addArgs functions.

```
addμ     : Exprμ → Termμ → Exprμ
addμ e t = ⟨ addIx , e :: t :: [] ⟩
```

```
data μ" (F : Fam) (t : TypeIx) : Set where
  <_>" : Σ (ConIx F t) (\c → Env (μ" F) (lookupCon F t c)) → μ" F t
```

In the next section we define a new module defining a separate interpretation function for each part of the encoding (Con, Type, and Fam). We do away with the typeInterp and lookupCon functions, but their functionality is still implemented. Semantically, the μ datatype is unchanged.

4.2.3 Interpretation module

We put the interpretation in a separate module and, as we did with the Codes module above, abstract over the number n of types in a family. We use the n to open the Codes module.

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

We also abstract out the interpretation function used by the Env. This abstraction enables us to define the interpretation functions without having to pass a Fam to each function and without having to use μ before we defined it.

For the interpretation of a value of Con (the encoding of constructor arguments) we use the environment type.

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

The function T⟦_⟧ implements the interpretation for Type.

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

The T⟦_⟧ function is similar to typeInterp but uses Type directly instead of looking the Type up in a Fam with a TypeIx. The lookup of Type is done in the interpretation function for Fam:

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


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

This definition of μ clearly shows how μ is used in its own definition to construct the interpretation function used by Env.

In Chapter 5 we use the Interpretation module. The example we use in that chapter illustrates how the above definitions are used and the full code of the example is in Appendix B.

4.3 Discussion

For our universe we do not follow the oft-used approach of using functors and sums-of-products. We chose a representation that closely corresponds to the labeled trees from Chapter 3. The result still is a sums-of-products approach – a family is a sum of types, which are products of constructors – however, our sums and products are of arbitrary arity. We do not need to consider the restrictions of binary structure and nesting.

In Chapter 5 we see that our view leads to a natural definition of Diff where ins, del, and cpy can refer to a value simply by using two indices: one for the type and one for the constructor. This view is therefore easier to work with in our case, than nested sum constructors and functors. A disadvantage is that we can't go deeper than this level: to represent constructors that again take products of sums we need to introduce new types. It is an inconvenience to come up with new labels, but we do not lose any expressiveness.

Another advantage we have over functors is that we can use our list indices to refer to other types. Using these references we can represent mutually recursive types without much hassle, similar to Multirec [27].

Chapter 5

Families

In this chapter we define generic, type-safe diffing and patching for families of datatypes. The algorithms resemble the diffing and patching algorithms for trees (Chapter 3). We use the codes from the universe we defined in the previous chapter to implement datatype-generic functions. The codes are also used to ensure the edit scripts are type-safe: it is not possible to define an edit script that creates an ill-typed value, as we did with the example in Section 3.5.

The running example in this chapter is a family with two mutually recursive datatypes:

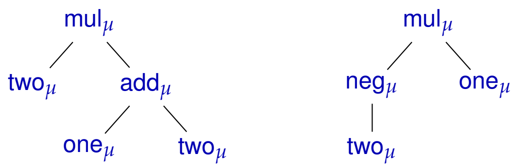
```
mutual
  data Expr : Set where
    add : Expr → Term → Expr
    one : Expr
  data Term : Set where
    mul : Term → Expr → Term
    neg : Term → Term
    two : Term
```

Using the Codes modules from Chapter 4 we encode the types and create readable names for the indices of the types and constructors (e.g., exprIx, mulIx, twoIx). With the Interpretation module, we define the types Expr_μ and Term_μ and functions isomorphic to the constructors of the datatypes above. The full code this example encoding and interpretation can be found in Appendix B.

$$
\begin{aligned}
&\mathsf{add}_\mu : \mathsf{Expr}_\mu \to \mathsf{Term}_\mu \to \mathsf{Expr}_\mu \\
&\mathsf{one}_\mu : \mathsf{Expr}_\mu \\
&\mathsf{mul}_\mu : \mathsf{Term}_\mu \to \mathsf{Expr}_\mu \to \mathsf{Term}_\mu \\
&\mathsf{neg}_\mu : \mathsf{Term}_\mu \to \mathsf{Term}_\mu \\
&\mathsf{two}_\mu : \mathsf{Term}_\mu
\end{aligned}
$$

5.1 Example

Before we look at the definition of the edit script, we first consider an example of how diffing two expressions works. We use the constructor functions defined above to write two example expressions, displayed as trees below:

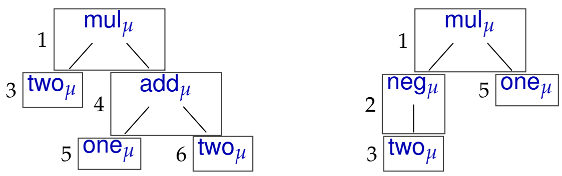


In the edit script for trees (Section 3.2.1), we used the label of the node as an argument to the edit script operations (the constructors ins, del, and cpy). The 'nodes' are now constructor functions, but we have an easy way to refer to them: we use the indices of the type and the constructor in the encoding. For example, add$_\mu$ is the constructor at position addIx in the type at position exprIx in the family, so we can refer to it with the pair (exprIx , addIx). When using an edit script with patch, we reconstruct the constructor functions using these indices.

The expressions above are relatively small, so we can write the edit script we expect the diff function to return:

```
1.  cpy (termIx , mulIx) $
2.  ins (termIx , negIx) $
3.  cpy (termIx , twoIx ) $
4.  del (exprIx , addIx) $
5.  cpy (exprIx , oneIx) $
6.  del (termIx , twoIx ) $
7.  end
```

The line numbers before each operation match the numbered boxes below. At step 1, all boxes annotated with 1 are consumed, at step 2 the box annotated with 2 is consumed, etc.



To prevent the construction of ill-typed edit scripts, we add extra type information to the Diff type. We parameterize the Diff type with the type indices of the encoding of the source and the target value. For example, the type of the edit script above is Diff (termIx :: []) (termIx :: []), because both the source and the target expression is of type Term$_\mu$. We use a list of type indices to represent a stack, to keep track of multiple subtrees (as we did with trees, in Section 3.2.2).

Each operation of the edit script has a recursive argument for the rest of the edit script of a parameterized type Diff. The exact type of the recursive argument depends on the operation since the parameters to the Diff type contain the type indices of the types currently on the stack. The type of each partial edit script for our example (starting at the corresponding line number above) is listed below:

```
1.  Diff (termIx :: [])          (termIx :: [])
2.  Diff (termIx :: exprIx :: []) (termIx :: exprIx :: [])
```

```
3.  Diff (termIx :: exprIx :: []) (termIx :: exprIx :: [])
4.  Diff (exprIx :: [])           (exprIx :: [])
5.  Diff (exprIx :: termIx :: []) (exprIx :: [])
6.  Diff (termIx :: [])           []
7.  Diff []                       []
```

The type of the complete edit script is at line 1. The first operation of the edit script is copying the mul_μ constructor. Because we copy, we consume the constructor from both the source and target, leaving two subexpressions. The following things happen to the type, in stack terms: we pop termIx from the list of type indices and push back the type indices of the arguments of the mul_μ constructor. The next operation is inserting the neg_μ constructor. We consume neg_μ from the target. The termIx type index is popped from the stack of type indices of the target. The argument of neg_μ is also of type Term_μ, so we push back termIx, ending up with the same type as the previous step. We continue until both expressions have been completely consumed and end with end of type Diff [] [] in step 7.

The pattern is straightforward: when we consume a constructor the type of the rest of the edit script changes. The type index of the constructor is removed from the head of the list of type indices, and the type indices of the arguments of the constructor are added at the front of the list. With our encoding, we can easily find the type indices of arguments of a constructor: using a type index and a constructor index we look up the matching Con.

In the next section we define the datatype for the edit script and formalize the pattern described above.

5.2 Edit script

The code of this chapter, excluding the examples, is grouped together in the module GenericDiff. The module is parameterized with the natural number n, the number of datatypes in the Family. We use n to open the Codes and Interpretation module from Chapter 4.

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

Inside this module, we create a module FamDiff that allows us to abstract over the family we define diffing and patching for and makes the family available as F inside the module.

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

The pair of the type index and the constructor index that we used in the example above is defined as a dependent pair:

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

The ConIx function uses the type index t to look up the type code in the family F, similar to the ConIx function of Section 4.2.2, but without the Fam argument, because we can now use the module parameter F.

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

The Diff datatype is parameterized by the TypeIxs of the source and the target. Each constructor in this definition of Diff has a different type, because we restrict how the source and target TypeIxs change.

```
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 []              []
```

For example, in the ins case, the recursive Diff argument must contain the fields of i on top its target stack. When applying the insert operation, these fields are used to construct a value of the type with typeix i, which is pushed on top of the stack without the fields (tys).

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

To illustrate why Diff helps us ensure type-safety, we revisit the example from Section 3.5, but use the types defined in this section. Consider the Diff

```
ins (exprIx , addIx) $ ins (termIx , twoIx) $ ins (termIx , twoIx) $ end
```

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

```
ins (termIx , twoIx) $ ins (termIx , twoIx) $ end
```

we can see that it is of type

```
Diff [] (termIx :: termIx :: [])
```

i.e., a Diff that creates two terms. 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.

5.3 Patching

We use the μEnv type constructor as a simple shorthand:

$$\begin{aligned} \mu\mathsf{Env} &: \quad \mathsf{TypeIxs} \to \mathsf{Set} \\ \mu\mathsf{Env} &= \mathsf{Env}\ (\mu\ \mathsf{F}) \end{aligned}$$

We apply Env to the interpretation function for the family the module is parameterized with.

The patch function uses the indices in the Diff type to calculate the type of the source and target value, by interpreting the indices using the environment for the family.

$$\begin{aligned} \mathsf{patch}\ :\ & \{\mathsf{txs}\ \mathsf{tys}\ :\ \mathsf{TypeIxs}\} \to \\ & \mathsf{Diff}\ \mathsf{txs}\ \mathsf{tys} \to \mu\mathsf{Env}\ \mathsf{txs} \to \mathsf{Maybe}\ (\mu\mathsf{Env}\ \mathsf{tys}) \end{aligned}$$

Note that despite the additional type information, patch is still partial. If we would want to assure patch always succeeds, we would have to parameterize the Diff datatype by not only the top-level type indices (from source and target), but by the indices of all the types and constructors in the expression. Including all type indices in the Diff type would lead to a trivial implementation of patch: we could easily project out the indices (of both the source and the target) and interpret them to values.

The definition of patch is again the same as the one for lists in Section 2.4, we only have to adapt the insert and delete functions.

5.3.1 Inserting

As the type of the ins constructor defines how the stacks of type indices must change, so does the type of the insert function:

$$\begin{aligned} \mathsf{insert}\ :\ & \{\mathsf{ts}\ :\ \mathsf{TypeIxs}\} \to (\mathsf{i}\ :\ \mathsf{Ixs}) \to \\ & \mu\mathsf{Env}\ (\mathsf{fields}\ \mathsf{i} +\!\!+ \mathsf{ts}) \to \mathsf{Maybe}\ (\mu\mathsf{Env}\ (\mathsf{typeix}\ \mathsf{i} :: \mathsf{ts})) \end{aligned}$$

The ins constructor only used the stacks of codes. In the definition of insert, however, we also use an environment. The type signature contains hints about what needs to happen in the implementation: using the encoding of the constructor (i), the environment contains values that match the types of the fields of the constructor and the rest (ts). If patching succeeds, we return an environment that starts with a value of the type encoded in i and leaves the rest (ts) unchanged.

$$\begin{aligned} \mathsf{insert}\ (\mathsf{t}, \mathsf{c})\ \mathsf{xss}\ =\ & \mathsf{splitEnv}\ (\mathsf{fields}\ (\mathsf{t}, \mathsf{c}))\ \mathsf{xss} \\ & (\lambda\ \mathsf{xs}\ \mathsf{ys} \to \mathsf{just}\ (\langle\ \mathsf{c}, \mathsf{xs}\ \rangle :: \mathsf{ys})) \end{aligned}$$

The function splitEnv (defined below) uses the type indices to split the environment in a part matching those type indices and the rest of the environment and passes them to a continuation function. In the continuation function we create a pair of the constructor index (c) and the fields (xs) and pass that to the constructor of μ to create the value, then prepend the value to the start of the environment.

Note how we can use the fields function in both the type and the definition. Also note that insert can not fail, because the type signature dictates that all required fields are present. Compare this insert function to the one we used for patching trees in Section 3.4, where we do not have the guarantee that the required *number* of children is present.

Splitting an environment

The splitEnv function splits a given environment Env I (txs ++ tys) into an Env I txs and an Env I tys, given a txs, 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)
```

As an example, consider again our simple universe of natural numbers and booleans, encoded with N and B, from Section 4.2.1. We defined the value environment:

```
environment : Env interpretation (N :: B :: B :: N :: [])
environment  =  4 :: false :: true :: 2 :: []
```

When we split the value above into two parts, we create a function that expects a continuation function that gets both parts as arguments.

```
splitEnvironment : {R : Set} →
   (Env interpretation (N :: B :: []) →
    Env interpretation (B :: N :: []) →
    R) → R
splitEnvironment  =  splitEnv (N :: B :: []) environment
```

5.3.2 Deleting

For deleting, we need the opposite function of splitEnv: a function that appends one environment to another. The +++ function is a straightforward implementation for environment concatenation, mimicking list concatenation (++).

```
_+++_ : {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)
```

Unlike the insert function, the delete function can fail: when the expected constructor index does not match the index of the constructor of the value, we return nothing.

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

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

We check if two constructor indices are equal with $\_ \overset{?}{=}_{\mathsf{Fin}} \_$, which maybe returns an equality proof.

```
_≟Fin_ : {n : ℕ} → (x y : Fin n) → Maybe (x ≡ y)
```

For the proof type, we use the propositional equality datatype from the Agda standard library:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

A value of x ≡ y encodes the equality of x and y. There is only one possible constructor, refl. If we pattern match on refl the type checker tries to infer that x and y are equal, if they are not equal, type checking fails. If type checking succeeds, the type checker can then use the information that x and y are equal in subsequent type checking. Note that we write .c in the case $c \overset{?}{=}_{\mathsf{Fin}} c'$ returns just refl. The dot tells Agda's type checker to first consider the rest of the expression. After it finds the refl constructor it can infer that the value at position .c is indeed equal to c.

The implementation of $\_ \overset{?}{=}_{\mathsf{Fin}} \_$ is straightforward.

```
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
```

5.4 Diffing

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 code for the algorithm is similar to the tree version from Chapter 3.

```
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))
```

Note that the type index stacks are passed implicitly, but are used to guide the pattern matching on the interpreted values. Agda checks the arguments of a function from left to right, and by pattern matching on the codes it infers the shape of the environment.

The diff function still has four cases, but the fourth one is split into two cases, depending on the equality of the type and constructor indices. The function $\_\stackrel{?}{=}_{\mathsf{Ixs}}\_$ performs the equality test and is implemented similarly as, and using, $\_\stackrel{?}{=}_{\mathsf{Fin}}\_$.

```
_≟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
```

The types of the edit script constructors restrict the implementation of diff, as expected: if we would try to write cpy as a possible solution when the type and constructor indices are different, the type checker makes sure the code does not compile.

5.5 Discussion

The implementation presented in this chapter is a working implementation. There are many opportunities for improvement though. For example, it is extremely slow. An efficient implementation is presented in Chapter 6.

While our universe can in theory represent types with very many or – using laziness – even infinite numbers of constructors such as Char and $\mathbb{N}$, it is clearly not efficient to use codes that way. Instead, it is desirable to represent such types in our encoded family using *abstract* types, types that do not contain an encoding of each constructor but use an existing implementation. We implement abstract types in Chapter 7.

To improve both readability of the edit scripts and efficiency of patching we can compress the Diff, merging cpy operations that copy complete subtrees. We implement compression in Chapter 8.

Chapter 6

Memoization

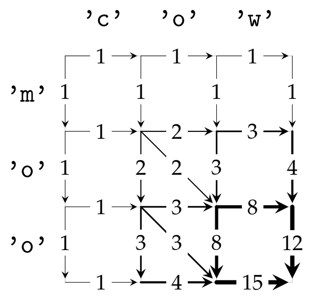
The implementations presented in the previous chapters are not very efficient. The algorithms calculate the best edit script, using the cost function, and therefore try every possible edit script, making two recursive calls or even three – when copying is possible – at each step. Many execution paths overlap, e.g., first deleting an item and then inserting another item results in the same subproblem as doing it vice versa.

To prevent recursive calls from repeatedly doing the same work, we need a mechanism to share subcomputations. If a particular subproblem is already computed, we want to reuse its solution. To enable this reuse, we need two things: a way to store subsolutions, and a way to identify subproblems so we can find the right solution.

In this chapter we show a memoization technique [18] that solves the sharing of subproblems. We first briefly discuss how memoization works with lists and then describe the adaptations we need to make to work with the generic algorithm we presented in Chapter 5.

6.1 Lists

To illustrate why the naïve list diffing algorithm is inefficient, consider the execution paths of diff with as input two lists of characters, `"cow"` and `"moo"`. We show the execution in a table. An arrow means a character is 'consumed'. We can consume a character from the source string (`"cow"`) by deleting, e.g., del `'c'` is depicted by a rightward arrow in the first column. Similarly, the ins operation is depicted by the downward arrows. The diagonal arrows depict the cpy operation; a character from both the source and target string is consumed.



The number of the arrow is number of times the execution path performed the operation depicted by that arrow. If we add the numbers of all arrows we calculate the number of steps the algorithm took: 81. If we simply count the number of arrows, we see there are only 21 different steps the algorithm can take. Due to the exponential nature of the algorithm duplicated execution paths dominate the execution time of the algorithm for any non-trivial example.

For lists, we can easily refer to subproblems by counting the number of characters consumed in both the source and target, e.g., the diff between `"ow"` and `"o"` can be referred to by `(1,2)`. In a imperative setting, a mutable nested array (table) is commonly used to save and look up subsolutions in. Note that each solution depends on its subsolution and therefore the table is filled starting at the 'empty' solution, in our example that is position `3,3`, corresponding with the lower right corner in the illustration above. When the cell at `0,0` is filled, we have a solution for the complete problem.

6.2 Trees

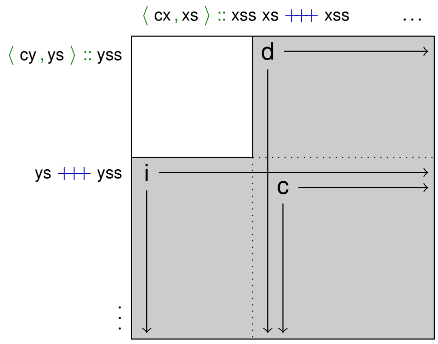
For our trees algorithm from Chapter 3 and also the generic algorithm from Chapter 5, we can also use a table. Because the trees are traversed with a depth-first preorder traversal, we have effectively serialized the problem: there is no extra branching in execution paths although we are dealing with branching structures. The same property still holds, deleting and then inserting a label or constructor still leads to the same subproblem as inserting first and then deleting.

In case of lists or tree diffing, the cells of the lookup table are all of the same type, namely Diff. In the generic case, however, our Diff has been parameterized by the lists of TypeIxs and each cell has a different type. We need to keep the type information also in the memoized algorithm, because when we look up the solution of a subproblem it still has to be of the correct type. Therefore, we define a custom data structure to build a table with a different type for each cell.

6.2.1 Table datatype

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

The last case is the most interesting, the situation is shown in the following picture:



At this point, there are three subproblems available, depending on the action we (can) take. If we delete cx the subproblem is d and the form of the source becomes xs +++ xss of type μEnv (fields (tx , cx) ++ txs). If we insert cy the subproblem is i and if cx and cy are the same constructor, we can copy and the subproblem is c.

The picture also shows that the i and d subproblems share the subproblem c, so even when we cannot copy we still have to calculate the c subproblem. Note that copying is a shortcut and deleting and then inserting (or vice versa) results in the same subproblem.

In the datatype we define to construct the table the cc constructor always has three subtables, containing the calculations for the representing the d, i and c subproblem.

The cn constructor represents the cells at the right border: the source has been consumed completely and only deleting is possible. Analogous, the nc constructor represents the cells at the bottom border, when only inserting is possible. Both the cn and nc constructor keep track of one subproblem table. The cell at the lower right is represented by the nn constructor, which does not have to track any subproblem.

```
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 []             []
```

Each constructor also contains the actual Diff representing the (best) solution for that cell. Having the Diff available at each cell allows us to easily extract the solution from a table using a simple function:

```
getDiff : ∀ {txs tys} → DiffT txs tys → Diff txs tys
getDiff (cc _ _ d _ _ _) = d
getDiff (cn _ d _)       = d
getDiff (nc _ d _)       = d
getDiff (nn d)           = d
```

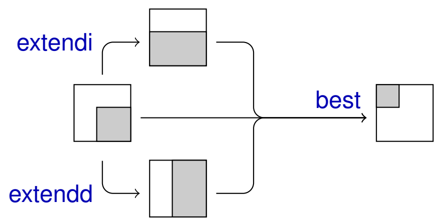
6.2.2 Diffing

For the memoized diffing, we write a function with the following signature:

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

The structure of the diffT function is very similar to the structure of diff. The cases where the source is empty (nc), the target is empty (cn), or both are empty (nn) are easy because they have no or only one recursive call. The main difference is in the cc case: instead of doing two or three recursive calls, we only make one recursive call to calculate the c (from the picture above) and use its result to extend the subsolution to the d and i. With all three subsolutions available, we chose the best result to extract the Diff from.

The following picture illustrates the idea behind the diffT function. The gray part of the block is the part of the table calculated by the function names shown as the labels of the arrows.



The code for diffT seems simpler than the diff code, but much of the complexity has moved to the extendi, extendd and best helper functions.

```
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
```

The last case clearly shows how c is shared and used to calculate the i and d.

The best function selects the best diff, similarly to the algorithm before. 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           )
```

In case the target is empty there are no rows in the table – we are at the final row, at the bottom of the table – so we can only have one subproblem table, thus we use the cn and extend the diff with a del operation. However, if there are rows in the table, we must add a cell to the left of each row, effectively extending the table with a column. Those cells are cc cells (except the last one), therefore we need three subtables, the d, i and c. The d was passed to the function. To get the c, we use the function extracti that drops a row from the d table. By extending that c table with a column (the recursive call), we get the i. Now we can build the cell and use the function best to pick the best diff of all subproblems.

The extracti function, which drops a row from the table, has a simple implementation. Since we store the subproblems at each cell, we can simply get the right subproblem. The type of the extracti function is more complex than its implementation and 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 extendd function (not shown) also has a extractd companion function, analoguous to extracti.

6.3 Discussion

Although we have presented the memoization code in Agda, we do not use this implementation in practice. Agda currently uses a call-by-name evaluation strategy when executing code. When using the Haskell backend this strategy causes the generated code in many cases to lose the carefully defined sharing. In Chapter 9, we look at the Haskell implementation of diffT that is a bit more involved, but does have a great performance increase compared to the Agda version. Also, Haskell's laziness helps to calculate only the parts of the table that are actually required to determine the result of the problem.

Chapter 7

Extension: Constants

As we note in the discussion of the generic solution (Section 5.5) a useful extension is to be able to use abstract types in the family of datatypes we use for diffing and patching. Abstract types are types for which we do not know or do not care to know the concrete structure. We do not consider the different constructors for that type, but rather deal with values 'as is'.

This chapter shows the code adaptations needed to encode abstract types in our solution. Not all code of the module is shown, many pieces, such as the Env, can remain unchanged. All code that has changed between modules *is* shown.

7.1 Codes

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.

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

The equality test is needed for the checks the diff and patch function perform when comparing constructors. The equality test for constructors has to be adapted which we show in Section 7.4.

Normally, a Set field is not allowed in a record of type Set, and we would have to give Abstract the type Set1, the type of Set. However, Agda has a flag to assume Set : Set, and we use it here to save the work of having to rewrite all our other definitions.

We make the distinction between abstract and concrete types in the encoding of types. By replacing the type synonym for Type with a datatype we can use pattern matching to find out if a type is concrete or abstract.

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

We illustrate the use of the new Type datatype with an example encoding of a family with two datatypes: the abstract Char type and a concrete list-like datatype for creating strings of characters.

First, we define readable names for the type indices.

```
charIx : TypeIx
charIx = zero

stringIx : TypeIx
stringIx = suc zero
```

We define the record for ‘char’ using the Char type and a function for decidable equality from the Data.Char library.

```
‘char’ : Type
‘char’ = abstr $ record {type = Char; decEq = _≟Char_}
```

The encoding of the string datatype is the same as before, with the small exception of having to use the concr constructor to create the Type from Lists of Cons.

```
‘nil’ : Con
‘nil’ = []

‘cons’ : Con
‘cons’ = charIx :: stringIx :: []

‘string’ : Type
‘string’ = concr $ ‘nil’ :: ‘cons’ :: []
```

The definition of the Fam type is not changed, so we put the two type encodings together in a vector to finish the encoding.

```
‘example’ : Fam
‘example’ = ‘char’ :: ‘string’ :: []
```

7.2 Interpretation

For the new Type code, we also need to adapt the interpretation function to handle both cases. We create an interpretation module as before (Section 4.2.3).

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

We define the interpretation of the new Type datatype also as a datatype

```
data T⟦_⟧ {I : TypeIx → Set} : Type → Set where
  «_» : {T : Abstract} → Abstract.type T → T⟦ abstr T ⟧
  _,_ : {T : List Con} → (c : Fin (length T))
        → C⟦ lookup c (fromList T) ⟧ I      → T⟦ concr T ⟧
```

The Type interpretation datatype is parametrized by the interpretation function. In the constructor for abstract types we store the type from the Abstract record. For concrete types we mimic the dependent pair we used before (Section 4.2.3). We also reuse the dependent pair constructor _ , _ so writing down interpretations for (concrete) types does not change.

The C⟦_⟧ and F⟦_⟧ functions, remain almost unchanged from the definitions in Section 4.2.3. Only the F⟦_⟧ function has to be slightly adapted because the interpretation function for TypeIx now has to passed as the first (implicit) argument.

As an example, we use the codes from the 'example' family defined earlier in this chapter to define the interpreted types.

```
Charμ   = μ 'example' charIx
Stringμ = μ 'example' stringIx
```

For the concrete 'string', the definitions look similar to those defined in Chapter 4.

```
nilμ : Stringμ
nilμ = ⟨ zero , [] ⟩
consμ : Charμ → Stringμ → Stringμ
consμ c cs = ⟨ suc zero , c :: cs :: [] ⟩
```

Note that we have not named the constructor indices for 'nil' and 'cons' but directly use values of Fin 2 (because there are two constructors in the 'string' encoding).

For abstract types we cannot create such functions as above, using only interpretations of encodings. We need to use values of the type we encoded, in this case Char, to construct values of the interpreted abstract type encoding.

```
charμ : Char → Charμ
charμ c = ⟨ « c » ⟩
```

The char_μ function is not really useful, as we can simply use Chars, e.g., to define a function to add a Char to a String_μ:

```
_::c_ : Char → Stringμ → Stringμ
_::c_ c cs = consμ ⟨ « c » ⟩ cs
infixr 5 _::c_
“moo„ = 'm' ::c 'o' ::c 'o' ::c nilμ
“cow„ = 'c' ::c 'o' ::c 'w' ::c nilμ
```

7.3 Edit script

The edit script has to undergo a couple of significant changes. We have to make the distinction between concrete and abstract types at several places, which we do by pattern matching on the constructors of the Type datatype. We start with opening a module and writing helper functions.

```
module GenericDiff+Constants (F : Fam) where
  ConType : Type → Set
```

```
ConType (concr T) = Fin (length T)
ConType (abstr T) = Abstract.type T

fields : (T : Type) → ConType T → TypeIxs
fields (concr T) c = lookup c (fromList T)
fields (abstr T) _ = []
```

The ConType function is similar to the ConIx function. However, for abstract types the constructor is not an index, but a value of the abstract type. The fields function for concrete types is as defined previously. For abstract type the result of fields is always an empty list, since the constructors are already values.

For the Diff, we can no longer use Idx, and because finding the type of the constructor index or value involves pattern matching, we split up the type index and constructor arguments. To do lookup only once, we use a let in the type.

```
data Diff : TypeIxs → TypeIxs → Set where
  ins  : {txs tys : TypeIxs}     (t : TypeIx)          →
         let T = lookup t F in   (c : ConType T)       →
         Diff txs                (fields T c ++ tys)   →
         Diff txs                (t :: tys)
  del  : {txs tys : TypeIxs} →   (t : TypeIx)          →
         let T = lookup t F in   (c : ConType T)       →
         Diff (fields T c ++ txs) tys                  →
         Diff (t :: txs)         tys
  cpy  : {txs tys : TypeIxs} →   (t : TypeIx)          →
         let T = lookup t F in   (c : ConType T)       →
         Diff (fields T c ++ txs) (fields T c ++ tys)  →
         Diff (t :: txs)         (t :: tys)
  end  : Diff []                 []
```

Since the Diff datatype changed, we must also adapt patching and diffing functions.

An example of the edit script, using the values “moo„ and “cow„ from the example above, is listed below

```
  cpy stringIx (suc zero)
$ del charIx 'm'
$ ins charIx 'c'
$ cpy stringIx (suc zero)
$ cpy charIx 'o'
$ cpy stringIx (suc zero)
$ del charIx 'o'
$ ins charIx 'w'
$ cpy stringIx zero
$ end
```

Note how all constructors of the String_μ type are copied and only the abstract characters change.

7.4 Patching

Previously, we could pattern match on the interpretations to get the constructor (index) and a list of its arguments. Creating a interpreted value was also trivial: we combined the constructor and the arguments in a pair and passed it to the fixed-point function. Because we now have to deal with two different cases, we abstract over applying and 'unapplying' constructors.

We apply both Env and the type interpretation function T⟦_⟧ to the interpretation function for the family this module was parameterized with (μ F), thereby creating two helper functions to deal with interpretations: one for interpreting a TypeIxs (similar to the result of fields) and one for interpreting a Type.

```
μEnv : TypeIxs → Set
μEnv = Env (μ F)
μT⟦_⟧ : Type → Set
μT⟦_⟧ = T⟦_⟧ {μ F}
```

To define the apply function we pattern match on the (implicit) Type argument to distinguish between abstract and concrete types.

```
apply : {T : Type} → (c : ConType T) → μEnv (fields T c) → μT⟦ T ⟧
apply {abstr _} c _  = « c »
apply {concr _} c ts = c , ts
```

For the 'unapply' we define a view [31]. A view in Agda consists of a datatype and a function. The datatype is used to define a view constructor, on which we can pattern match to get our information, in this case the constructor and the arguments. The function, unapply, takes the information the view needs and constructs it.

```
data Unapply : (T : Type) → μT⟦ T ⟧ → Set where
  _,_ : {T : Type} → (c : ConType T) → (ts : μEnv (fields T c)) →
        Unapply T (apply c ts)
unapply : (t : TypeIx) → (e : μT⟦ lookup t F ⟧) → Unapply (lookup t F) e
unapply t e with lookup t F
unapply t (c , args) | concr _ = c , args
unapply t (« c »)    | abstr _ = c , []
```

The type of patch stays the same

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

and the implementation only changes to incorporate the split of the type index and constructor arguments

```
patch (ins t c d) ys = (insert t c ◇ patch d             ) ys
patch (del t c d) ys = (               patch d ◇ delete t c) ys
patch (cpy t c d) ys = (insert t c ◇ patch d ◇ delete t c) ys
patch end         []  = just []
```

The insert function caters to the new helper function, but most notably it uses apply instead of the _,_ constructor.

```
insert : {ts : TypeIxs} → (t : TypeIx) →
         let T = lookup t F in (c : ConType T) → μEnv (fields T c ++ ts) →
         Maybe (μEnv (t :: ts))
insert t c xss = splitEnv (fields (lookup t F) c) xss
                    (λ xs ys → just (⟨ apply c xs ⟩ :: ys))
```

For delete, we need a new function to test the equality between constructors.

```
_||_≟_ : (T : Type) → (cx cy : ConType T) → Maybe (cx ≡ cy)
(concr _) || cx ≟ cy   = cx ≟Fin cy
(abstr T) || cx ≟ cy  with Abstract.decEq T cx cy
(abstr T) || cx ≟ .cx | yes refl = just refl
...                   | no _     = nothing
```

Again, we have to pattern match on a Type to be able to use the correct function to test for equality.

We use unapply in a **with** pattern to be able to pattern match on the interpreted value (we write ._, because unapply provides us with the values). The equality test for the constructors uses the function defined previously.

```
delete : {ts : TypeIxs} → (t : TypeIx) →
         let T = lookup t F in (c : ConType T) → μEnv (t :: ts) →
         Maybe (μEnv (fields T c ++ ts))
delete t c (⟨ e ⟩ :: xss) with unapply t e
delete t c (⟨ ._ ⟩ :: xss) | c′ , xs with lookup t F || c ≟ c′
delete t c (⟨ ._ ⟩ :: xss) | .c , xs | just refl = just (xs +++ xss)
delete t c (⟨ ._ ⟩ :: xss) | c′ , xs | nothing   = nothing
```

7.5 Diffing

The diff function becomes a bit more verbose: the unapply is used in a **with** pattern. Because the result of the unapply determines the interpretation, we need to repeat it in the line below and cannot shorten it with '...'. Another factor is that we no longer can check if the constructors are the same, because we first compare the types. We repeat the definition unequal constructors twice: once for different types and once for equal types.

```
diff : ∀ {txs tys} → μEnv txs → μEnv tys → Diff txs tys
diff {[]}       {[]}       []              []              =
     end
diff {tx :: _} {[]}        (⟨ ex ⟩ :: xss) []              with unapply tx ex
diff {tx :: _} {[]}        (⟨ ._ ⟩ :: xss) []              | cx , xs =
     del tx   cx (diff (xs +++ xss) [])
diff {[]}       {ty :: _} []              (⟨ ey ⟩ :: yss) with unapply ty ey
diff {[]}       {ty :: _} []              (⟨ ._ ⟩ :: yss) | cy , ys =
     ins ty   cy (diff [] (ys +++ yss))
diff {tx :: _} {ty :: _} (⟨ ex ⟩ :: xss) (⟨ ey ⟩ :: yss)
```

```
   with unapply tx ex | unapply ty ey
diff {tx :: _} {ty :: _} (⟨ ._ ⟩ :: xss) (⟨ ._ ⟩ :: yss)
  | cx , xs         | cy , ys with tx ≟Fin ty
...                      | nothing =
     ins ty cy    (diff (⟨ apply cx xs ⟩ :: xss) (ys ++ yss))
   ⊓ del tx cx    (diff (xs ++ xss) (⟨ apply cy ys ⟩ :: yss))
diff {tx :: _} {.tx :: _} (⟨ ._ ⟩ :: xss) (⟨ ._ ⟩ :: yss)
  | cx , xs         | cy , ys | just refl with lookup tx F || cx ≟ cy
...                                  | nothing =
     ins tx cy    (diff (⟨ apply cx xs ⟩ :: xss) (ys ++ yss))
   ⊓ del tx cx    (diff (xs ++ xss) (⟨ apply cy ys ⟩ :: yss))
diff {tx :: _} {.tx :: _} (⟨ ._ ⟩ :: xss) (⟨ ._ ⟩ :: yss)
  | cx , xs         | .cx , ys | just refl | just refl =
     ins tx cx    (diff (⟨ apply cx xs ⟩ :: xss) (ys ++ yss))
   ⊓ del tx cx    (diff (xs ++ xss) (⟨ apply cx ys ⟩ :: yss))
   ⊓ cpy tx cx    (diff (xs ++ xss) (ys ++ yss))
```

While the diff is less good looking now, adding constants to our solution makes it closer to being useful in practice. In the Haskell version (Chapter 9) we have a slightly different solution for constants due to the way types are encoded.

7.6 Discussion

It is important that the interpretation of the Type code datatype is also defined as a datatype. If we use a function, we lose the distinction between concrete and abstract types at this point and run into trouble later when trying to pattern match on interpretation values.

Chapter 8

Extension: Compression

When comparing two data structures we are often mostly interested in the differences. For example, in practice, when comparing two revisions of a source code file, the number of differences is relatively small compared to the amount of code that remains unchanged. The output format of UNIX's `diff` leaves out the parts that are unchanged, except for some lines around the changes. Those lines offer a context to UNIX's `patch` and help it find the lines that must be changed even if the source file is not exactly the same as the one used to calculate the edit script.

The goal of the compression extension described in this chapter is to create smaller edit scripts by compressing parts that are the same in both the source and target value. It is not necessary to leave some parts as the context: our edit scripts are type-safe, so we have guarantees that the right parts are changed. Furthermore, our patch function does not have the heuristics to offer the robustness against small changes in the source value that UNIX's `patch` does. A more robust patch function is discussed in Section 10.1, future work.

The code in this chapter is an adaptation of the code presented in Chapter 5. While we can also apply the compression to the patches presented in Chapter 7, we do not incorporate the constants extension, to keep the code simpler. In the Haskell version in Chapter 9 we show a version of the algorithms using all extensions.

We implement compression by creating an alternative Diff datatype and do the compression as a post-processing step on the result of diff. This way, we do not have to adapt the diff algorithm.

8.1 Example

Consider again the example family used in Section 5.1; the full encoding of this family can be found in Appendix B.

If we call diff on the source expression $\mathsf{mul}_\mu\ \mathsf{two}_\mu\ \mathsf{one}_\mu$ and target expression $\mathsf{neg}_\mu\ (\mathsf{mul}_\mu\ \mathsf{two}_\mu\ \mathsf{one}_\mu)$ the result is the following edit script:

```
ins (termIx , negIx) $
cpy (termIx , mulIx) $
cpy (termIx , twoIx) $
```

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

Since the mul two_μ one_μ part of the expression did not change, we have an opportunity for compression, as we can copy this complete subexpression, replacing three cpy operations by one cpyAll operation:

```
ins     (termIx , negIx) $
cpyAll termIx            $
end
```

The information the cpyAll operation needs takes the TypeIx of the encoding of the type as an argument. We do not save the constructor index, it is not necessary.

8.2 Edit Script

To support compression we need to extend the edit script datatype with the cpyAll operation.

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

The type of the cpyAll construct is simpler than the other constructors, since we do not need any type functions, e.g. to refer to the fields.

8.3 Compressing

The compress function takes a Diff and returns a Diff of the same type as a result.

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

The edit script is compressed recursively. In case we find a cpy operation we first compress the rest of the edit script and then use the copied function to check if the result of the recursive call also was compression. The result of the copied function is either nothing or the Diff with all operation for the subtree stripped off, so we can replace them with a cpyAll operation.

```
compress (cpy i d) with compress d
...                | d′ with copied (fields i) d′
... | nothing  = cpy i d′
... | just d″  = cpyAll (typeix i) d″
```

In all other cases we simply continue recursively with the compression.

```
compress (del i d)    = del i    (compress d)
compress (ins i d)    = ins i    (compress d)
```

```
compress (cpyAll t d)  =  cpyAll t (compress d)
compress end           =  end
```

The copied function uses a similar technique as the splitEnv function from Section 5.3.1: we pass a list of type indices for the fields which we use to check exactly the part of the Diff that contains the operations for those fields.

```
copied : {txs tys : TypeIxs} → (tzs : TypeIxs) →
             Diff (tzs ++ txs) (tzs ++ tys) → Maybe (Diff txs tys)
copied []          ds              =  just ds
copied (tz :: tzs) (cpyAll .tz d)  =  copied tzs d
copied (tz :: tzs) _               =  nothing
```

Because compression happens recursively before the check with copied is performed, we only need to check for cpyAll. A subexpression can only be compressed if all its arguments are also compressed.

8.4 Patching and diffing

For patching, the type can stay unchanged but we need to add a case for the cpyAll constructor.

```
patch : {txs tys : TypeIxs} →
              Diff txs tys → μEnv txs → Maybe (μEnv tys)
patch ... [] = just []
patch (cpyAll t d) (y :: ys)  =  _::_ y ⟨$⟩ patch d ys
```

The code is simpler than for the normal operations, since we do not get the fields from the interpretation and do not check constructors for equality.

The ⟨$⟩ operator is a functor operation. For this specific case it can also be defined as:

```
_⟨$⟩_ : {A B : Set} → (A → B) → Maybe A → Maybe B
f ⟨$⟩ (just x)  =  just (f x)
_ ⟨$⟩ nothing   =  nothing
```

For diffing, we do not need to adapt the algorithm itself, but do have to add the cpyAll constructor to the cost function. This addition is only needed to complete the definition of cost, the diff function does not produce cpyAll operations.

```
cost : {txs tys : TypeIxs} → Diff txs tys → ℕ
cost ...  =  0
cost (cpyAll _ d)  =  1 + cost d
```

8.5 Discussion

The compression functionality showed in this chapter is only a start. This work can be extended in several ways, e.g., integrating the compression with the diffing (which is not necessary if we use it in a lazy language).

We can also extend the compression to ins and del. If we want to compress ins operation, however, the insAll constructor must take an interpreted value as an argument, which causes the edit script no longer to consist of only simple indices.

The compression used by the UNIX `diff` for edit scripts requires `patch` to search for the correct lines to patch. We do not have that 'problem,' but our (uncompressed) edit scripts are also not as flexible and cannot deal with input slightly different than expected. The compression of cpy operations also make the edit script more flexible. We do not store or check if the constructor matches, but only look at the type. The patch function using compressed edit scripts is therefore more forgiving if the source has changed slightly, but still ensures type safety.

Chapter 9

Haskell implementation

In this chapter we show the Haskell implementation of the algorithms presented in the previous chapters. Haskell is not as suitable for programming with types as Agda, and the techniques we use to do programming with dependent types in Haskell make the code a bit more complex than the code of the previous chapters. Another factor increasing the complexity of the Haskell code is that we combine the work of all previous chapters into a single implementation.

The reason we create a Haskell implementation is that it makes the algorithms useful in practice: we can compile it to efficient executable code and we can use available libraries to offer datatypes, e.g., abstract syntax, to define specific instances of our algorithms. By making our algorithms available as a library too, they can also be used in other applications.

9.1 Universe

The universe in Haskell differs from the universe we used in Agda. In Haskell we cannot calculate types from codes, so we keep the types as a part of the codes, using a GADT. Having the types available in the encoding also makes our interpretation easier, as we can recreate values of the exact type. In Agda, our interpretation of the codes was isomorphic to the datatypes, not identical.

We explain the steps and types involved in building the universe with an example. Our family consists of two datatypes we define ourselves and of Int:

```
data Expr = Min Expr Term
data Term = Parens Expr
          | Number Int
```

First, we define a GADT that captures the structure of the family. Each constructor in the family is encoded as a constructor in the GADT. The GADT for the family looks as follows:

```
data ExampleFamily :: * → * → * where
  'Min'    ::        ExampleFamily Expr (Cons Expr (Cons Term Nil))
  'Parens' ::        ExampleFamily Term (Cons Expr Nil)
  'Number' ::        ExampleFamily Term (Cons Int Nil)
  'Int'    :: Int → ExampleFamily Int Nil
```

We use type level lists to capture the types of the fields of each constructor, using two separate datatypes as the constructors.

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

As Int is an abstract type in the family, we define the 'Int' code as a constructor that expects an actual value.

The next step is to make the GADT an instance of a class Family that provides several generic functions that we can need to define the generic diff algorithm. The Family class is defined as:

```
class Family f where
  decEq :: f tx txs → f ty tys → Maybe (tx:=:ty, txs:=:tys)
  fields  :: f t ts → t → Maybe ts
  apply   :: f t ts → ts → t
  string  :: f t ts → String
```

The decEq corresponds to the $\overset{?}{=}_{\mathsf{Ixs}}$ function from Section 5.4; however, we do not compare indices, but actual types. The proofs we return use the equality GADT:

```
data a:=:b where
  Refl :: a:=:a
```

The definition of :=: assures that if we write Refl, a and b are of the same type, or else the type checker will complain.

The fields function tries to match an encoded constructor with the actual constructor (of the same type). If it succeeds, it returns the fields of the matched constructor.

The inverse function of fields is apply, which applies the actual constructor, given an encoding of that constructor, to a list of fields.

The string function is a simple show for the GADT, allowing us to show a string representation for each constructor, to be able to inspect edit scripts.

Defining the instance of Family for ExampleFamily is straightforward.

```
instance Family ExampleFamily where
  decEq 'Min'      'Min'      = Just (Refl, Refl)
  decEq 'Parens'   'Parens'   = Just (Refl, Refl)
  decEq 'Number'   'Number'   = Just (Refl, Refl)
  decEq ('Int' x)  ('Int' y)  | x == y    = Just (Refl, Refl)
                              | otherwise = Nothing
  decEq _ _                   = Nothing
  fields 'Min'     (Min e t)    = Just (Cons e (Cons t Nil))
  fields 'Parens'  (Parens e)   = Just (Cons e Nil)
  fields 'Number'  (Number i)   = Just (Cons i Nil)
  fields ('Int' _) _            = Just Nil
  fields _         _            = Nothing
  apply 'Min'      (Cons e (Cons t Nil)) = Min e t
  apply 'Parens'   (Cons e Nil)          = Parens e
  apply 'Number'   (Cons i Nil)          = Number i
```

```
apply ('Int' i)     Nil          = i
string 'Min'      = "Min"
string 'Parens'   = "Parens"
string 'Number'   = "Number"
string ('Int' i)  = show i
```

The cases for handling the abstract Int type are different from the rest. The decEq function also needs to check the actual value, the fields function always matches, the apply function extracts the value and the string function uses the normal show.

The third and last step in encoding a family in our universe is to create an instance of the class Type for each type in family. Using the type class Type allows us to get just the constructors for a specific type from the family GADT.

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

In the Agda version, we used the Fam type, which is a vector of type encodings. Using a TypeIx we can get all the constructors of a certain type. Because we now put all constructors together in the same GADT, we need the Type class to be able to separate them again.

The datatype Con wraps the representation GADT such that the type of the fields of the constructor is hidden, so we can put the fields together in a (normal) list.

```
data Con :: (∗ → ∗ → ∗) → ∗ → ∗ where
  Concr :: (List f ts)      ⇒        f t ts  → Con f t
  Abstr :: (Eq t, List f ts) ⇒ (t → f t ts) → Con f t
```

Here we also make the separation between concrete and abstract types. The Concr constructor is used for concrete encodings, the Abstr constructor packs a function that expects a value of the encoded type and wraps it into an encoding for family f.

```
instance Type ExampleFamily Term where
  constructors = [Concr 'Number', Concr 'Parens']
instance Type ExampleFamily Expr where
  constructors = [Concr 'Min']
instance Type ExampleFamily Int where
  constructors = [Abstr 'Int']
```

Note that for abstract types, the constructors function always is a singleton with an Abstr wrapping the encoding constructor from the family GADT.

The List class used in the types of Con's constructors restricts the value of ts to a list of Nil and Cons containing only elements that are types in the family f.

```
class List f ts where
  list :: IsList f ts
```

List uses the IsList GADT as the type of its only function.

```
data IsList :: (* → * → *) → * → * where
  IsNil  ::                          IsList f Nil
  IsCons :: (Type f t) ⇒ IsList f ts → IsList f (Cons t ts)
```

IsList has two constructors that restrict the types ts to the type list constructors, Nil and Cons. The List class has two instances, one for each 'constructor':

```
instance List f Nil where
  list = IsNil
instance (Type f t, List f ts) ⇒ List f (Cons t ts) where
  list = IsCons list
```

The List class and IsList datatype mimic the Env datatype we defined in Section 4.2.1 of the Agda implementation

Note that all generic functions in the Haskell library are parameterized by the family. In Agda, we can use a parameterized module to make the family available to all functions (in that module). Unfortunately, parameterized modules are not supported in Haskell, so we have to pass the family around explicitly.

As a real-life example of how to use the universe defined above, we created a JSON [11] example in Appendix C

9.2 Edit script

The Diff datatype looks very similar to the Agda version. We do need to use several class constraints to assure the types working with Diff can pattern match on the heterogeneous lists and detect the constructors for the given type.

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

Note that we also made CpyTree available in Diff, to be able to compress the Diff.

The Append type is a type function to concatenate type-level lists, encoded in Haskell as 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)
```

To concatenate the values of lists we have two functions, using Append. The appendList function builds a new IsList value. The append function uses the IsList constructors to do pattern matching on the actual values to be append.

```
appendList :: IsList f txs → IsList f tys → IsList f (Append txs tys)
appendList IsNil isys = isys
appendList (IsCons isxs) isys = IsCons (appendList isxs isys)

append :: IsList f txs → IsList f tys → txs → tys → Append txs tys
append IsNil _ Nil ys = ys
append (IsCons isxs) isys (Cons x xs) ys = Cons x (append isxs isys xs ys)
```

Note that we cannot pattern match on values of the List class unless we also have the IsList value for that list.

Using the string function from the Family class, we can now define a simple instance of Show for the edit script, so we can print it for inspection.

```
instance Show (Diff f txs tys) where
  show (Ins  c    d) = "Ins " ++ string c ++ " $ " ++ show d
  show (Del  c    d) = "Del " ++ string c ++ " $ " ++ show d
  show (Cpy  c    d) = "Cpy " ++ string c ++ " $ " ++ show d
  show (CpyTree d)   = "CpyTree"          ++ " $ " ++ show d
  show End           = "End"
```

9.3 Patching

Not only the definition of the edit script, but also the definition of the patch function in Haskell is very similar to the definition in Agda, presented in Chapter 2.

```
patch :: ∀f txs tys.Diff f txs tys → txs → tys
patch (Ins  c    d) = insert c ∘ patch d
patch (Del  c    d) =            patch d ∘ delete c
patch (Cpy  c    d) = insert c ∘ patch d ∘ delete c
patch (CpyTree d)   = λ(Cons x xs) → Cons x ∘ patch d $ xs
```

Note that the patch function is partial, as is the patch function in Agda, but we do not use a Maybe to catch an invalid patch but rather throw an exception, either because the pattern matching fails, or explicitly, as demonstrated in the delete function:

```
delete :: (Type f t, List f ts, List f txs) ⇒ f t ts → Cons t txs → Append ts txs
delete c (Cons x xs) =
  case fields c x of
    Nothing → error "Patching failed"
    Just ts → append (isList c) list ts xs
```

The fields function from the Family class checks if the value matches the encoding of the constructor in c.

The isList function is a helper that provides easy access to the IsList value of the fields list of the argument passed, which we need for appending.

```
isList :: (Family f, List f ts) ⇒ f t ts → IsList f ts
isList _ = list
```

In the insert case we use the apply function to get the constructor function from the encoding and apply it to the encoded fields list.

```
insert :: (Type f t, List f ts, List f txs) ⇒ f t ts → Append ts txs → Cons t txs
insert c xs = Cons (apply c txs) tys
  where (txs, tys) = split (isList c) xs
```

The split function is used to split the fields for the constructor off the stack, as splitEnv did in the Agda implementation in Section 5.3.1

```
split :: IsList f txs → Append txs tys → (txs, tys)
split IsNil ys                      = (Nil, ys)
split (IsCons isxs) (Cons x xsys) = let (xs, ys) = split isxs xsys
                                    in  (Cons x xs, ys)
```

9.4 Diffing

In contrast to the patch function above, defining diff in Haskell requires a bit more effort. The two main reasons for that are that we need to use IsList to be able to do pattern matching and that we need to define a function that matches the encoding with the actual value.

We use the fields function of the type class Family in combination with the constructors function of the type class Type to restrict the encodings we need to try when matching. We still have to iterate over all possible constructors in a type, though.

The matchConstructor function takes a value of type t and a continuation that is applied to the representation f t cs of the constructor that matches t and the fields of that constructor ts.

```
matchConstructor :: (Type f t) ⇒ t →
                    (∀ts.f t ts → IsList f ts → ts → r) → r
matchConstructor = tryEach constructors
  where
    tryEach :: (Type f t) ⇒ [Con f t] → t →
      (∀ts.f t ts → IsList f ts → ts → r) → r
    tryEach (Concr c : cs) x k = matchOrRetry c     cs x k
    tryEach (Abstr c  : cs) x k = matchOrRetry (c x) cs x k
    tryEach [] _ _ = error "Incorrect Family or Type instance."
    matchOrRetry :: (List f ts, Type f t) ⇒ f t ts → [Con f t] → t →
      (∀ts.f t ts → IsList f ts → ts → r) → r
    matchOrRetry c cs x k = case fields c x of
      Just ts   → k c (isList c) ts
      Nothing → tryEach cs x k
```

We show the implementation of the diff function to introduce several concepts we use in the Haskell implementation. In practice, this definition of diff is not used in favor of the more efficient diffT implementation which is also shown, in Section 9.6.

The type of the diff function is still simple, but its implementation relies on the IsList constructors to guide the pattern matching on the source and target lists and is given in the function diff'.

```
diff :: (Family f, List f txs, List f tys) ⇒ txs → tys → Diff f txs tys
diff = diff' list list
```

The cases where both or either the source and target lists are empty are relatively simple.

```
diff' :: (Family f) ⇒ IsList f txs → IsList f tys → txs → tys → Diff f txs tys
diff' IsNil          IsNil          Nil          Nil          =
    End
diff' (IsCons isxs) IsNil          (Cons x xs) Nil          =
  matchConstructor x
    (λcx isxs' xs' →
    del isxs' isxs cx
      (diff' (appendList isxs' isxs)    IsNil
             (append isxs' isxs xs' xs) Nil))
diff' IsNil          (IsCons isys) Nil          (Cons y ys) =
  matchConstructor y
    (λcy isys' ys' →
    ins isys' isys cy
      (diff' IsNil (appendList isys' isys)
             Nil   (append isys' isys ys' ys)))
```

Note the functions del and ins that are used instead of the constructors Del and Ins. The del and ins functions use the IsList values to reify the type classes the Del and Ins constructor require. Their implementation is explained at the end of this section.

Next, we look at the case when both source and target contain items. We already calculate the recursive cases, but delegate the decision for the best operation to bestDiff.

```
diff' (IsCons isxs) (IsCons isys) (Cons x xs) (Cons y ys) =
  matchConstructor x
    (λcx isxs' xs' →
      matchConstructor y
      (λcy isys' ys' →
        let c = diff' (appendList isxs' isxs)    (appendList isys' isys)
                      (append isxs' isxs xs' xs) (append isys' isys ys' ys)
            d = diff' (appendList isxs' isxs)    (IsCons isys)
                      (append isxs' isxs xs' xs) (Cons y ys)
            i = diff' (IsCons isxs)              (appendList isys' isys)
                      (Cons x xs)                (append isys' isys ys' ys)
        in bestDiff cx cy isxs' isxs isys' isys i d c))
```

Lazyness allows us to already write the recursive diff' for when we can copy, but we do not have to worry about it being executed unless we need it.

Finally, the bestDiff function uses the information of the decEq function to decide whether the cpy operation is applicable and uses best to pick the best solution.

```
bestDiff :: (Type f tx, Type f ty) ⇒ f tx txs' → f ty tys' →
  IsList f txs' → IsList f txs → IsList f tys' → IsList f tys →
  Diff f (Cons tx txs)       (Append tys' tys) →
  Diff f (Append txs' txs) (Cons ty tys) →
  Diff f (Append txs' txs) (Append tys' tys) →
  Diff f (Cons tx txs)       (Cons ty tys)
bestDiff cx cy isxs' isxs isys' isys i d c = case decEq cx cy of
  Just (Refl, Refl) → best (cpy isxs' isxs isys cx c) $
                       best (del  isxs' isxs cx d)
                            (ins  isys' isys cy i)
  Nothing            → best (del  isxs' isxs cx d)
                            (ins  isys' isys cy i)
```

The best function returns the shortest edit script. We calculate the length of the diff as a Peano natural, Nat, to be able to use lazy comparison.

```
best :: Diff f txs tys → Diff f txs tys → Diff f txs tys
best dx dy = bestSteps (steps dx) dx (steps dy) dy

data Nat = Zero | Succ Nat
  deriving (Eq, Show)

steps :: Diff f txs tys → Nat
steps (Ins  _ d) = Succ $ steps d
steps (Del  _ d) = Succ $ steps d
steps (Cpy _ d) = Succ $ steps d
steps End         = Zero

bestSteps :: Nat → d → Nat → d → d
bestSteps Zero x _       _ = x
bestSteps _       _ Zero y = y
bestSteps (Succ nx) x (Succ ny) y = bestSteps nx x ny y
```

Note that we do not include a case for CpyTree in steps. Unlike Agda, Haskell allows us to leave out cases when implementing a function. This feature allows us to be more succinct in the cases when we can easily outsmart the compiler.

The final piece of the puzzle is to define the ins, del and cpy functions. These functions exist to get rid of the IsList witnesses and reinstantiate the List class constraint for the constructors of the edit script.

We define the datatype RList. Its only constructor, RList, uses the List class constraint. Companioned by the function reify, which turns an IsList into an RList, we can convince Haskell's type checker that the List class constraint is applicable.

```
data RList :: (∗ → ∗ → ∗) → ∗ → ∗ where
  RList :: List f ts ⇒ RList f ts

reify :: IsList f ts → RList f ts
reify IsNil            = RList
reify (IsCons ists) = case reify ists of
                         RList → RList
```

The actual definitions of ins, del and cpy are not interesting, so we only show ins:

```
ins :: (Type f t) ⇒ IsList f ts → IsList f tys →
  f t ts → Diff f txs (Append ts tys) → Diff f txs (Cons t tys)
ins ists isys =
  case (reify ists, reify isys) of
    (RList, RList) → Ins
```

We also use the trick to provide functions for the constructors of the table datatype used by diffT in Section 9.6, on memoization.

9.5 Compression

The algorithm for compression, replacing expressions that are fully Cpy'd by a CpyTree, is very similar to the one in Agda, defined in Chapter 8.

```
compress :: (Family f) ⇒ Diff f txs tys → Diff f txs tys
compress End           = End
compress (Ins  c    d) = Ins c     (compress d)
compress (Del  c    d) = Del c     (compress d)
compress (CpyTree d)   = CpyTree (compress d)
compress (Cpy c     d) = let d' = compress d in
  case copied (isList c) d' of
    Just d''  → CpyTree d''
    Nothing → Cpy c d'
copied :: (Family f) ⇒ IsList f ts →
  Diff f (Append ts txs) (Append ts tys) → Maybe (Diff f txs tys)
copied IsNil               d      = Just d
copied (IsCons xs) (CpyTree d)    = copied xs d
copied (IsCons _)  _              = Nothing
```

9.6 Memoization

The implementation of memoization in Haskell does not add new concepts, neither compared to the code above nor to the definition from Agda in Chapter 6. We highlight a few subtle differences, but mostly show the code for completeness.

9.6.1 Table datatype

The DiffT datatype describes the memoization table.

```
data DiffT :: (∗ → ∗ → ∗) → ∗ → ∗ → ∗ where
  CC :: (Type f tx, Type f ty, List f txs', List f tys') ⇒
    f tx txs' → f ty tys' →
    Diff f (Cons tx txs) (Cons ty tys) →
    DiffT f (Cons tx txs)      (Append tys' tys) →
```

```
      DiffT f (Append txs' txs) (Cons ty tys)       →
      DiffT f (Append txs' txs) (Append tys' tys) →
      DiffT f (Cons tx txs)      (Cons ty tys)
    CN :: (Type f tx, List f txs') ⇒ f tx txs' →
      Diff f (Cons tx txs) Nil →
      DiffT f (Append txs' txs) Nil →
      DiffT f (Cons tx txs)      Nil
    NC :: (Type f ty, List f tys') ⇒ f ty tys' →
      Diff f Nil (Cons ty tys) →
      DiffT f Nil                (Append tys' tys) →
      DiffT f Nil                (Cons ty tys)
    NN :: Diff f Nil Nil →
      DiffT f Nil                Nil
```

9.6.2 Diffing

The diffT function calculates the DiffT table

```
diffT :: (Family f, List f txs, List f tys) ⇒ txs → tys → DiffT f txs tys
diffT = diffT' list list
diffT' :: (Family f) ⇒ ∀txs tys.IsList f txs → IsList f tys →
  txs → tys → DiffT f txs tys
diffT' IsNil          IsNil          Nil          Nil          =
  NN End
diffT' (IsCons isxs) IsNil          (Cons x xs) Nil          =
  matchConstructor x
    (λcx isxs' xs' →
      let d = diffT' (appendList isxs' isxs)      IsNil
                     (append isxs' isxs xs' xs) Nil
      in cn isxs' isxs cx (del isxs' isxs cx (getDiff d)) d)
diffT' IsNil          (IsCons isys) Nil          (Cons y ys) =
  matchConstructor y
    (λcy isys' ys' →
      let i = diffT'  IsNil (appendList isys' isys)
                      Nil   (append isys' isys ys' ys)
      in nc isys' isys cy (ins isys' isys cy (getDiff i)) i)
diffT' (IsCons isxs) (IsCons isys) (Cons x xs) (Cons y ys) =
  matchConstructor x
    (λcx isxs' xs' →
      matchConstructor y
      (λcy isys' ys' →
        let c = diffT' (appendList isxs' isxs)      (appendList isys' isys)
                       (append isxs' isxs xs' xs) (append isys' isys ys' ys)
            d = extendd isys' isys cy c
            i  = extendi  isxs' isxs cx c
        in cc isxs' isxs isys' isys cx cy
             (bestDiffT cx cy isxs' isxs isys' isys i d c) i d c))
```

from which the resulting Diff can be extracted:

```
getDiff :: DiffT f txs tys → Diff f txs tys
getDiff (CC _ _ d _ _ _) = d
getDiff (CN _ d _) = d
getDiff (NC _ d _) = d
getDiff (NN d)     = d
```

The bestDiffT function is similar to the bestDiff function and selects the best Diff from the three recursive solutions.

```
bestDiffT :: (Type f tx, Type f ty) ⇒ f tx txs' → f ty tys' →
             IsList f txs' → IsList f txs → IsList f tys' → IsList f tys →
             DiffT f (Cons tx txs)      (Append tys' tys) →
             DiffT f (Append txs' txs) (Cons ty tys)      →
             DiffT f (Append txs' txs) (Append tys' tys) →
             Diff f (Cons tx txs) (Cons ty tys)
bestDiffT cx cy isxs' isxs isys' isys i d c = case decEq cx cy of
  Just (Refl, Refl) → cpy isxs' isxs isys cx (getDiff c)
  Nothing → best (ins isys' isys cy (getDiff i))
                 (del isxs' isxs cx (getDiff d))
```

The function extendi (and similarly, the function extendd) use pattern matching on the table datatype. In Agda, we used pattern matching on the lists, but Haskell's type checker does not allow that, as pattern matching makes the type txs or txs' 'rigid'.

```
extendi :: (Type f tx) ⇒ IsList f txs' → IsList f txs → f tx txs' →
  DiffT f (Append txs' txs) tys' →
  DiffT f (Cons tx txs)      tys'
extendi isxs' isxs cx dt@(NN d)       = cn isxs' isxs cx (del isxs' isxs cx d) dt
extendi isxs' isxs cx dt@(CN _ d _) = cn isxs' isxs cx (del isxs' isxs cx d) dt
extendi isxs' isxs cx dt@(NC _ _ _) = extendi' isxs' isxs cx dt
extendi isxs' isxs cx dt@(CC _ _ _ _ _ _) = extendi' isxs' isxs cx dt

extendi' :: (Type f tx, Type f ty) ⇒ IsList f txs' → IsList f txs → f tx txs' →
  DiffT f (Append txs' txs) (Cons ty tys) →
  DiffT f (Cons tx txs)      (Cons ty tys)
extendi' isxs' isxs cx dt =
  extracti dt (λisys' isys cy dt' →
    let i = extendi isxs' isxs cx dt'
        d = dt
        c = dt'
    in cc isxs' isxs isys' isys cx cy
          (bestDiffT cx cy isxs' isxs isys' isys i d c)
          i d c)
```

From the Agda implementation we know that we only need to implement two cases for extracti:

```
extracti :: (Type f ty) ⇒ DiffT f txs' (Cons ty tys) →
  (∀tys'.IsList f tys' → IsList f tys → f ty tys' →
  DiffT f txs' (Append tys' tys) → r) → r
extracti (CC _ c d i _ _) k = k (isList c) (targetTail d) c i
extracti (NC c d i) k = k (isList c) (targetTail d) c i
```

The targetTail helper function retrieves the IsList of the extracted value.

```
targetTail :: Diff f txs (Cons ty tys) → IsList f tys
targetTail (Ins _ d) = list
targetTail (Del _ d) = targetTail d
targetTail (Cpy _ _) = list
```

For nc, cn and cc we use the trick with RList and reify once more:

```
nc :: (Type f t) ⇒ IsList f ts → IsList f tys →
  f t ts → Diff f Nil (Cons t tys) →
  DiffT f Nil (Append ts tys) → DiffT f Nil (Cons t tys)
nc ists isys =
  case (reify ists, reify isys) of
    (RList, RList) → NC
```

9.7 Discussion

Looking at the example from the start of the chapter and also Appendix C we can see there is a lot of boilerplate code that needs to be written in order to instantiate the encoding of the family. Because all the code is straightforward, it could be automatically generated given the syntax tree of the datatype definitions, using a preprocessor or a meta-programming library such as Template Haskell [29].

A shortcoming of the definitions presented in this chapter is that the family is closed, because we defined its encoding in a single datatype. We can not, as we can with the Agda definitions, where a family encoding is a vector, easily take an existing family encoding and (programmatically) extend it. Furthermore, we can not encode polymorphic datatypes such as lists, but need to write specific encodings for each type we want to include.

Chapter 10

Conclusion

In this thesis I have presented how to go from a simple algorithm for diffing lists to an algorithm for trees that we subsequently used for the main contribution of this thesis: diffing and patching datatypes generically while ensuring type-safety. I showed a few extensions to the algorithm and demonstrated how the algorithms are implemented in Agda and Haskell. Another major contribution is that we implemented memoization for our type-safe algorithms. This required us to build a special memoization table datatype where not only the value in a cell but also the type of a cell depends on other cells.

10.1 Related and future work

The only work (of which we are aware) that comes close to being a generic diff in a functional programming language is from Piponi [25, 26] 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.

There are several directions in which this work can be extended. An interesting direction is to more closely look at the work by Chawathe and Garcia-Molina on meaningful change detection in structured data [9]. Their work has a different goal, more focused on the semantics of changes, which leads to completely different trees, edit scripts and a heuristic algorithm to find the edit script. The algorithm has to be heuristic, since the trees are unordered and calculating the difference between two unordered trees is NP-Hard.

There are several questions that come to mind

- How do we define a typed edit script with operations such as swapping, moving and updating? What is the minimal set of primitive operations that can be used to express these operations?
- Can we implement the heuristic algorithm with the new, but still typed, edit script? Will the extra type information hinder the implementation or help it?
- Can we support partially unordered trees? Datatypes are inherently ordered, but for some parts the ordering might not be important. For ex-

ample, in many programming languages lists are ordered, but dictionaries are not.

For the minimal set of primitive operations, the work on lenses [12] by Foster et al. [12] might be interesting. Lenses are combinators for bi-directional tree transformations. Using a (combined) lens you can create a 'view' on piece of data such that changes to that view can be translated back to the original data. In a sense, interpreting a plain text file as structured data is taking a 'view' on it. In our examples we did ignore all whitespace, but if we could use lenses and calculate our patches on the 'view', we might be able to translate the patched results back to plain text without loss of formatting.

On the practical side, there are also several aspects that need work. Making the algorithms presented more suitable to be used as a library and reducing the work a programmer has to do to be able to use this work is largely a software engineering problem, but might also bring to light more fundamental problems. The library could also be integrated in an application, e.g., a structured editor, a version control system or a command line tool.

If we want to transfer the edit scripts, we need to be able to serialize them to disk and read them while preserving the types. To a certain extent the deserialization can be done using a simple parser, but the use of dependent types might make reconstructing the value a non-trivial problem.

10.2 Acknowledgements

First and foremost, I would like to thank you, dear reader, for reading this thesis. While writing a Master's Thesis is an interesting and important exercise, I do not have the illusion that a thesis is in general well-read, if read at all. Even if you did not find what you are looking for, your interest in my work means a lot to me.

I did not expect to learn so much in a year without taking any classes. Andres has taught me a lot and I very much enjoyed working with him. I admire his ability to clearly explain topics, making difficult things sound natural and logical. He was not afraid to call my work poor, when it was, but also repeatedly expressed his trust in my capabilities.

Sean's calmness, perfectionism and (native) English skills impressed me. Even under a looming deadline he keeps his cool and manages to deliver excellent work. Although he was less involved than Andres in the daily matters, he provided great help and without him my talks and this thesis would have been less successful.

I thank Chris for being patient with me and covering for me by working harder in and on our company at the times I was in crunch mode for this thesis or a talk. I look forward to doing the same for him.

Last but not least, I thank the most patient, trusting and loving of all: Didy. She is the absolute best at simply being there for me, which was all I needed.

Appendix A

Agda syntax for Haskellites

Agda [20] is a dependently typed programming language, using an extension of Martin-Löf's type theory. Agda's syntax bears resemblance to Haskell [24], making it easy to understand for people familiar with Haskell, but there are subtle and not so subtle differences to keep in mind. This appendix lists the most important differences and is intended to be both a fast introduction and a reference for Agda (syntax) for Haskell programmers.

A.1 UTF-8

Agda fully supports UTF-8 and allows almost all characters to be used in identifiers. For instance, the type for natural numbers, is ℕ, one can write both **forall** and ∀, etc. Most code in this thesis is therefore not the result of some fancy formatting, but simply shows the richness of UTF-8.

Because almost all characters are allowed in identifiers, it is always necessary to add spaces around operators. Parentheses and curly braces are special, and cannot be used in identifiers and therefore also do not need extra spacing.

A.2 Colons

In Agda, an value identifier is separated from its type by a single colon.

```
fib : ℕ → ℕ
fib 0 = 1
fib 1 = 1
fib n = fib (n ∸ 1) + fib (n ∸ 2)
```

Double colons are (often) list constructors. There is no special syntax for lists, only the two constructors [] and ::.

```
tail : ∀ {A} → List A → List A
tail []       = []
tail (x :: xs) = xs
```

We have to write the ∀ {A}, in Haskell that is implied. Notice the curly braces? Those are explained next.

A.3 Implicit arguments

Implicit arguments are written within curly braces. Without implicit arguments, we define the function above as:

```
tail : ∀ A → List A → List A
tail A []         = []
tail A (x :: xs)  = xs
```

The type argument for the list is now explicit, so we also need to pattern match on it. The use of a type variable this way might be a bit mind boggling, since in Haskell we can not mix types and values this way. In Agda we can!

A.4 Kinds and named type arguments

In Haskell the type of a type is called a *kind* and we only have one: $\star$. In Agda, $\star$ is called Set, and we still call it a type.

In the above example, the type checker restricts A to Set, because List takes an argument of type Set. We could also have written:

```
tail : {A : Set} → List A → List A
tail []         = []
tail (x :: xs)  = xs
```

The signature now reads as: given an implicit argument A of type Set and a List of A, we produce a List of A. Note how we *name* the Set argument A, so that we can use it for both Lists.

A.5 Underscores: infix, mixfix

Underscores are used as a pattern match wildcard (as in Haskell), but also in identifiers, to indicate where arguments go. For example, an operator (infix function) is defined as:

```
_+_ : ℕ → ℕ → ℕ
zero   + n = n
suc m + n = suc (m + n)
```

The reason Agda uses underscores when defining operators is that not only infix function, but also postfix and even 'mixfix' function can be defined using underscores. For example:

```
if_then_else_ : {A : Set} → Bool → A → A → A
if true  then t else f = t
if false then t else f = f
```

A.6 Constructors

Constructors in Agda are usually written in lowercase, but that is not a restriction. For example: just, nothing, true, false.

Constructors can be overloaded. For example, the constructors used for lists

```
data List (A : Set) : Set where
  []   : List A
  _::_ : (x : A) (xs : List A) → List A
```

are also used for vectors:

```
data Vec (A : Set) : ℕ → Set where
  []   : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)
```

While overloading might seem confusing, it is generally useful in practice because it is not necessary to create unique constructors for each datatype, which might not even be used together. Agda can almost always infer the type of the constructors from the context.

A.7 Dependent types

The Vec type above is a classic example of dependently typed programming. The type of Vec depends on a value, a natural number representing the length of the vector.

```
data Vec (A : Set) : ℕ → Set where
  []   : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)
```

Not only simple values are allowed in types, we can program in them just as we normally do with values. For example:

```
concat : ∀ {A m n} → Vec (Vec A m) n → Vec A (n ⋆ m)
concat []          = []
concat (xs :: xss) = xs ++ concat xss
```

A.8 with syntax

Agda does not have guards but does have the **with** syntax, which is similar to Haskell's **case ... of**

```
takeWhile : ∀ {A} → (A → Bool) → List A → List A
takeWhile p []        = []
takeWhile p (x :: xs) with p x
takeWhile p (x :: xs) | true  = x :: takeWhile p xs
takeWhile p (x :: xs) | false = []
```

Using **with** allows pattern matching on the result of a function call within a function definition.

It is also possible to chain several **with**s, simply by separating each pattern match with a bar (| |).

A.8.1 ...

Instead of repeating the left hand side of the **with**, we can also use the shorthand .... The above definition can be rewritten as:

```
takeWhile : ∀ {A} → (A → Bool) → List A → List A
takeWhile p []         = []
takeWhile p (x :: xs) with p x
...                    | true   = x :: takeWhile p xs
...                    | false  = []
```

In some cases we need to write the left hand side, because the **with** pattern match gives us more information about variables in the left hand side. See the next section for more details.

A.9 Fin

```
data Fin : ℕ → Set where
  zero : {n : ℕ} →             Fin (suc n)
  suc  : {n : ℕ} (i : Fin n) → Fin (suc n)
```

Fin n is a type with exactly n members. It is not possible to use Fin to define a type with no members (Fin 0), because the successor of the (implicit) argument n is used as the type argument. Since n is a natural number, Fin 1 is the type with only one member: zero {0}. The suc constructor adds a given member of Fin n to the members of Fin (suc n). Therefore, Fin 2 can only have two members: zero {1} and suc {1} (zero {0}).

The Fin n type is very useful to limit indices for lookups in structures of size n, for example vectors.

Appendix B

Example datatype encoding

This appendix contains the full code for the example used in Chapter 5. It lists the codes used to encode the family and functions created using the interpretation functions from Chapter 4.

B.1 Family

The family consists of two mutually-recursive datatypes.

```
mutual
  data Expr : Set where
    add : Expr → Term → Expr
    one : Expr
  data Term : Set where
    mul : Term → Expr → Term
    neg : Term → Term
    two : Term
```

B.2 Codes

We open the Codes module parameterized with 2 (the number of datatypes in the family we want to encode) and define the type indices and encodings for the constructors, types and the family.

```
open Codes 2
```

B.2.1 Type indices

```
exprIx : TypeIx
exprIx = zero

termIx : TypeIx
termIx = suc zero
```

B.2.2 Constructor encodings

```
'add' : Con
'add' = exprIx :: termIx :: []
'one' : Con
'one' = []
'neg' : Con
'neg' = termIx :: []
'mul' : Con
'mul' = termIx :: exprIx :: []
'two' : Con
'two' = []
```

B.2.3 Type encodings

```
'expr' : Type
'expr' = 'add' :: 'one' :: []
'term' : Type
'term' = 'mul' :: 'neg' :: 'two' :: []
```

B.2.4 Family encoding

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

B.3 Interpretation

We open the Interpretation module with the same natural number, 2, as the Codes module. We define the types and constructor functions for our interpreted codes, isomorphic to the constructors of the datatypes we encoded.

```
open Interpretation 2
```

B.3.1 Types

```
Exprμ : Set
Exprμ = μ 'example' exprIx
Termμ : Set
Termμ = μ 'example' termIx
```

B.3.2 Constructor indices

Using a simple helper function to construct the type from the type index, we create the constructor indices with readable names.

```
ConIx : TypeIx → Set
ConIx t = Fin (length (lookup t ‘example’))
addIx : ConIx exprIx
addIx = zero
oneIx : ConIx exprIx
oneIx = suc zero
mulIx : ConIx termIx
mulIx = zero
negIx : ConIx termIx
negIx = suc zero
twoIx : ConIx termIx
twoIx = suc (suc zero)
```

B.3.3 Constructor functions

```
addμ : Exprμ → Termμ → Exprμ
addμ e t = ⟨ addIx , e :: t :: [] ⟩
oneμ : Exprμ
oneμ = ⟨ oneIx , [] ⟩
mulμ : Termμ → Exprμ → Termμ
mulμ t e = ⟨ mulIx , t :: e :: [] ⟩
negμ : Termμ → Termμ
negμ t = ⟨ negIx , t :: [] ⟩
twoμ : Termμ
twoμ = ⟨ twoIx , [] ⟩
```

Appendix C

Haskell example: JSON

We use a JSON [11] library from Hackage to provide the datatypes for the abstract syntax of JSON data.

```
import Text.JSON
import Text.JSON.String
```

Using the universe defined in Section 9.1 we define all necessary datatypes and functions to be able to use the Haskell implementation of diff and patch from Chapter 9.

C.1 Family GADT

```
data JSONFam :: * → * → * where
  'Bool'        :: Bool     → JSONFam Bool   Nil
  'Rational'    :: Rational → JSONFam Rational Nil
  'String'      :: String   → JSONFam String Nil
  '[]'          :: JSONFam [JSValue] Nil
  '(:)'         :: JSONFam [JSValue] (Cons JSValue (Cons [JSValue] Nil))
  '[](,)'       :: JSONFam [(String, JSValue)] Nil
  '(:)(,)'      :: JSONFam [(String, JSValue)]
                     (Cons (String, JSValue)
                           (Cons [(String, JSValue)] Nil))
  (,)           :: JSONFam (String, JSValue)
                     (Cons String (Cons JSValue Nil))
  'JSONString'  :: JSString → JSONFam JSString Nil
  'JSNull'      :: JSONFam JSValue Nil
  'JSBool'      :: JSONFam JSValue (Cons Bool Nil)
  'JSRational'  :: JSONFam JSValue (Cons Bool (Cons Rational Nil))
  'JSString'    :: JSONFam JSValue (Cons JSString Nil)
  'JSArray'     :: JSONFam JSValue (Cons [JSValue] Nil)
  'JSObject'    :: JSONFam JSValue (Cons (JSObject JSValue) Nil)
  'JSONObject'  :: JSONFam (JSObject JSValue)
                     (Cons [(String, JSValue)] Nil)
```

C.2 Family instance

```
instance Family JSONFam where
  decEq ('Bool' x)        ('Bool' y)        | x == y    = Just (Refl, Refl)
                                            | otherwise = Nothing
  decEq ('Rational' x)    ('Rational' y)    | x == y    = Just (Refl, Refl)
                                            | otherwise = Nothing
  decEq ('String' x)      ('String' y)      | x == y    = Just (Refl, Refl)
                                            | otherwise = Nothing
  decEq '[]'              '[]'              = Just (Refl, Refl)
  decEq '(:)'             '(:)'             = Just (Refl, Refl)
  decEq '[](,)'           '[](,)'           = Just (Refl, Refl)
  decEq '(:)(,)'          '(:)(,)'          = Just (Refl, Refl)
  decEq (,)               (,)               = Just (Refl, Refl)
  decEq ('JSONString' x) ('JSONString' y) | x == y    = Just (Refl, Refl)
                                            | otherwise = Nothing
  decEq 'JSNull'          'JSNull'          = Just (Refl, Refl)
  decEq 'JSBool'          'JSBool'          = Just (Refl, Refl)
  decEq 'JSRational'      'JSRational'      = Just (Refl, Refl)
  decEq 'JSString'        'JSString'        = Just (Refl, Refl)
  decEq 'JSArray'         'JSArray'         = Just (Refl, Refl)
  decEq 'JSObject'        'JSObject'        = Just (Refl, Refl)
  decEq 'JSONObject'      'JSONObject'      = Just (Refl, Refl)
  decEq _                 _                 = Nothing
  fields ('Bool' _)         _                = Just Nil
  fields ('Rational' _)     _                = Just Nil
  fields ('String' _)       _                = Just Nil
  fields '[]'               []               = Just Nil
  fields '(:)'              (x : xs)         = Just (Cons x (Cons xs Nil))
  fields '[](,)'            []               = Just Nil
  fields '(:)(,)'           (x : xs)         = Just (Cons x (Cons xs Nil))
  fields (,)                (x, y)           = Just (Cons x (Cons y Nil))
  fields ('JSONString' _)   _                = Just Nil
  fields 'JSNull'           JSNull           = Just Nil
  fields 'JSBool'           (JSBool x)       = Just (Cons x Nil)
  fields 'JSRational'       (JSRational x y) = Just (Cons x (Cons y Nil))
  fields 'JSString'         (JSString x)     = Just (Cons x Nil)
  fields 'JSArray'          (JSArray x)      = Just (Cons x Nil)
  fields 'JSObject'         (JSObject x)     = Just (Cons x Nil)
  fields 'JSONObject'       x                = Just (Cons (fromJSObject x)
                                                      Nil)
  fields _                  _                = Nothing
  apply ('Bool' x)        Nil                      = x
  apply ('Rational' x)    Nil                      = x
  apply ('String' x)      Nil                      = x
  apply '[]'              Nil                      = []
  apply '(:)'             (Cons x (Cons xs Nil))   = x : xs
  apply '[](,)'           Nil                      = []
  apply '(:)(,)'          (Cons x (Cons xs Nil))   = x : xs
```

```
apply (,)                (Cons x (Cons y Nil))  = (x, y)
apply ('JSONString' x)   Nil                    = x
apply 'JSNull'           Nil                    = JSNull
apply 'JSBool'           (Cons x Nil)           = JSBool x
apply 'JSRational'       (Cons x (Cons y Nil))  = JSRational x y
apply 'JSString'         (Cons x Nil)           = JSString x
apply 'JSArray'          (Cons x Nil)           = JSArray x
apply 'JSObject'         (Cons x Nil)           = JSObject x
apply 'JSONObject'       (Cons x Nil)           = toJSObject x

string ('Bool' x)        = show x
string ('Rational' x)    = show x
string ('String' x)      = show x
string '[]'              = "[]"
string '(:)'             = "(:)"
string '[](,)'           = "[]"
string '(:)(,)'          = "(:)"
string (,)               = "(,)"
string ('JSONString' x)  = show x
string 'JSNull'          = "JSNull"
string 'JSBool'          = "JSBool"
string 'JSRational'      = "JSRational"
string 'JSString'        = "JSString"
string 'JSArray'         = "JSArray"
string 'JSObject'        = "JSObject"
string 'JSONObject'      = "JSONObject"
```

C.3 Type instances

```
instance Type JSONFam Bool where
  constructors = [Abstr 'Bool']
instance Type JSONFam Rational where
  constructors = [Abstr 'Rational']
instance Type JSONFam String where
  constructors = [Abstr 'String']
instance Type JSONFam [JSValue] where
  constructors = [Concr '[]', Concr '(:)']
instance Type JSONFam [(String, JSValue)] where
  constructors = [Concr '[](,)', Concr '(:)(,)']
instance Type JSONFam (String, JSValue) where
  constructors = [Concr (,)]
instance Type JSONFam JSString where
  constructors = [Abstr 'JSONString']
instance Type JSONFam JSValue where
  constructors = [Concr 'JSNull', Concr 'JSBool', Concr 'JSString'
                 , Concr 'JSArray', Concr 'JSObject']
```

```
instance Type JSONFam (JSObject JSValue) where
  constructors = [Concr 'JSONObject']
```

C.4 Example

As an example, we look at the following two JSON files.

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

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

Using UNIX's `diff` command to find the difference between the two files, we get the following output:

```
@@ -1,3 +1,3 @@
 [ "foo",
-  [ "bar",
-    "baz" ] ]
+  "bar",
+  "baz" ]
```

To use the type-safe generic diffing, we parse the JSON files and get the following output for the source (left) file

```
JSArray [JSString (JSONString {fromJSString = "foo"}),
         JSArray [JSString (JSONString {fromJSString = "bar"}),
                  JSString (JSONString {fromJSString = "baz"})]]
```

and the target (right) file.

```
JSArray [JSString (JSONString {fromJSString = "foo"}),
         JSString (JSONString {fromJSString = "bar"}),
         JSString (JSONString {fromJSString = "baz"})]
```

The result of the diff (or actually, a diffT and a getDiff) is a value equal to the following edit script:

```
  Cpy 'JSArray'
$ Cpy '(:)'
$ Cpy 'JSString'
$ Cpy ('JSONString' $ toJSString "foo")
$ Cpy '(:)'
$ Del 'JSArray'
$ Del '(:)'
$ Cpy 'JSString'
$ Cpy ('JSONString' $ toJSString "bar")
$ Cpy '(:)'
$ Cpy 'JSString'
$ Cpy ('JSONString' $ toJSString "baz")
```

```
$ Cpy '[]'
$ Del  '[]'
$ End
```

If we run compress on the edit script above the result is the following, much smaller, edit script:

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

Bibliography

[1] Bazaar. URL http://bazaar-vcs.org.

[2] Darcs. URL http://darcs.net.

[3] Git. URL http://git.or.cz.

[4] Mercurial. URL http://www.selenic.com/mercurial.

[5] Subversion. URL http://subversion.tigris.org.

[6] M. Benke, P. Dybjer, and P. Jansson. Universes for Generic Programs and Proofs in Dependent Type Theory. *Nordic Journal of Computing*, 10(4): 265–289, 2003.

[7] 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.

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

[9] S. S. Chawathe and H. 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.

[10] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. 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.

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

[12] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, May 2007. Preliminary version presented at the *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004; extended abstract presented at *Principles of Programming Languages (POPL)*, 2005.

[13] J. Gibbons. Datatype-Generic Programming. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Datatype-Generic Programming*, pages 1–71. Springer Berlin/Heidelberg, 2007.

[14] D. S. Hirschberg. *The longest common subsequence problem*. PhD thesis, Princeton, NJ, USA, 1975.

[15] S. Holdermans, J. Jeuring, A. Löh, and A. Rodriguez. Generic Views on Data Types. In T. Uustalu, editor, *MPC 2006: Proceedings of the 8th International Conference on the Mathematics of Program Construction*, pages 209–234. July 2006.

[16] P. 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.

[17] A. Lozano and G. 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.

[18] D. Michie. "memo" functions and machine learning. *Nature*, 218(5136): 19–22, April 1968.

[19] P. Morris. *Constructing Universes for Generic Programming*. PhD thesis, The University of Nottingham, November 2007.

[20] U. Norell. Dependently typed programming in agda. URL `http://www.cs.chalmers.se/~{}ulfn/darcs/AFP08/LectureNotes/AgdaIntro.pdf`.

[21] B. C. D. S. Oliveira, R. Hinze, and A. Löh. Extensible and Modular Generics for the Masses. In H. Nilsson, editor, *Trends in Functional Programming*, volume 7 of *Trends in Functional Programming*, pages 199–216. Intellect, 2006.

[22] N. Oury and W. 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.

[23] L. 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.

[24] S. L. Peyton Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98: A non-strict, purely functional language. Technical report, feb 1999. URL `http://www.haskell.org/definition/`.

[25] D. Piponi. The Antidiagonal, September 2007. URL `http://blog.sigfpe.com/2007/09/type-of-distinct-pairs.html`.

[26] D. Piponi. Tries and their Derivatives, September 2007. URL `http://blog.sigfpe.com/2007/09/tries-and-their-derivatives_08.html`.

[27] A. Rodriguez, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *Accepted to ICFP 2009*, 2009.

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

[29] T. Sheard and S. P. Jones. Template Meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002.

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

[31] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 307–313. ACM Press, 1987. ISBN 0897912152.

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

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