Universiteit Utrecht

Faculty of Science

Dept. of Information and Computing Sciences

Generic diffing and merging of mutually recursive datatypes in Haskell

Author

Arian van Putten (4133935)

Supervisor

Dr. Wouter Swierstra

Dr. Alejandro Serrano Mena

Daily Supervisor

Victor Cacciari Miraldo

May 14, 2019

In liefdevolle herrinering aan mijn papa, die altijd zo trots op mij was.

Contents

1 Introduction

1.1 Motivation

Version control systems are an important tool for the collaboration of software developers. They allow multiple people to work simultaneously on the same code repository effectively. It is the task of the version control system to reconcile changes when multiple people edit the same piece of a file. There are multiple techniques employed by different version control systems to do this reconciliation. The most widely-used software-merging tools are unstructured in nature. Popular tools are the `diff` and `diff3` tools, which power version control systems like subversion and git. Unstructured approaches are simple. These systems treat all code as lines of text, and use the *largest common sub-sequences* algorithm to find changes on a line-by-line basis [**lcs**]. Though this approach gives us a performant approach for finding changes between files, reconciling these changes can be tricky at times. When two people make changes to the same line of code that seem to be independent `diff3` will still find a conflict, because each line of code is treated as a separate character in the diffing algorithm. Sometimes single units of change might also span multiple lines of code, instead of a subset of lines, and in this case `diff3` might also struggle during reconciliation.

As an illustrating example, we have picked a merge conflict from the popular Lua project Kong [1]. Two changes were performed in parallel on same same file by two different people. The first contributor created a commit that refactored some parts of the code in a predictable way. Namely, replacing all occurrences of the string `ngx.` with `ngx_`.

```
@@ -237,7 +241,7 @@ function Kong.init_worker()
     return "init"
   end)
   if not ok then
-    ngx.log(ngx.CRIT, "could not set router version in cache: ", err)
+    ngx_log(ngx_CRIT, "could not set router version in cache: ", err)
     return
   end

@@ -291,7 +295,7 @@ function Kong.balancer()

     local ok, err = balancer_execute(addr)
     if not ok then
-      ngx.log(ngx.ERR, "failed to retry the dns/balancer resolver for ",
+      ngx_log(ngx_ERR, "failed to retry the dns/balancer resolver for ",
               addr.upstream.host, "' with: ", tostring(err))
```

[1] `https://github.com/Kong/kong`

```
        return responses.send(500)
```

At the same time, another person fixed a bug in the logging code. The field `addr.upstream.host` ceised to exist some versions ago, which caused the logger to always print `nil`. The person came up with the following simple patch to fix that:

```
@@ -292,7 +292,7 @@ function Kong.balancer()
     local ok, err = balancer_execute(addr)
     if not ok then
       ngx.log(ngx.ERR, "failed to retry the dns/balancer resolver for ",
-              addr.upstream.host, "' with: ", tostring(err))
+              tostring(addr.host), "' with: ", tostring(err))

       return responses.send(500)
     end
```

Though these changes are obviously disjoint, `diff3` will report a conflict, because a single line was changed in two different ways at the same time.

```
    local ok, err = balancer_execute(addr)
    if not ok then
<<<<<<< A.lua
      ngx_log(ngx_ERR, "failed to retry the dns/balancer resolver for ",
              addr.upstream.host, "' with: ", tostring(err))
=======
      ngx.log(ngx.ERR, "failed to retry the dns/balancer resolver for ",
              tostring(addr.host), "' with: ", tostring(err))
>>>>>>> B.lua

      return responses.send(500)
    end
```

1.2 Approach

Structured syntactic merging tries to reduce the amount of conflicts that occur, by exploiting the underlying structure of the file that is being diffed. This can be accomplished by parsing the files that we want to diff into abstract syntax tree. In Figure 1.1, we again see the changes that were made to the Lua file, but this time the patches are represented as changes over the abstract syntax tree of the changed files. In these images, the color red indicates a deletion of part of the tree. This means that the tree on which we apply the diff, must have all the components marked in red, and these are then removed to produce the result of the diff. White parts denote parts of the tree that must be preserved when applying the patch. They have to be present in the source tree when applying the patch, and will be copied over verbatim. Green parts of the patch denote insertions. These are new parts that will be added to the tree when we apply the patch. Finally in yellow we have `Scp` nodes. These nodes match any value in the tree on which we are applying the patch, and copies whatever it matches over to the new tree. The existence of these `Scp` nodes makes our patches applicable to a wide domain of trees, instead of limiting the domain of a patch to just a single tree.

If we look carefully at the `Scp` nodes in both patches, we see that the two patches are actually *disjoint*. They both operate on the source tree *O.lua independently*. At every place where the two

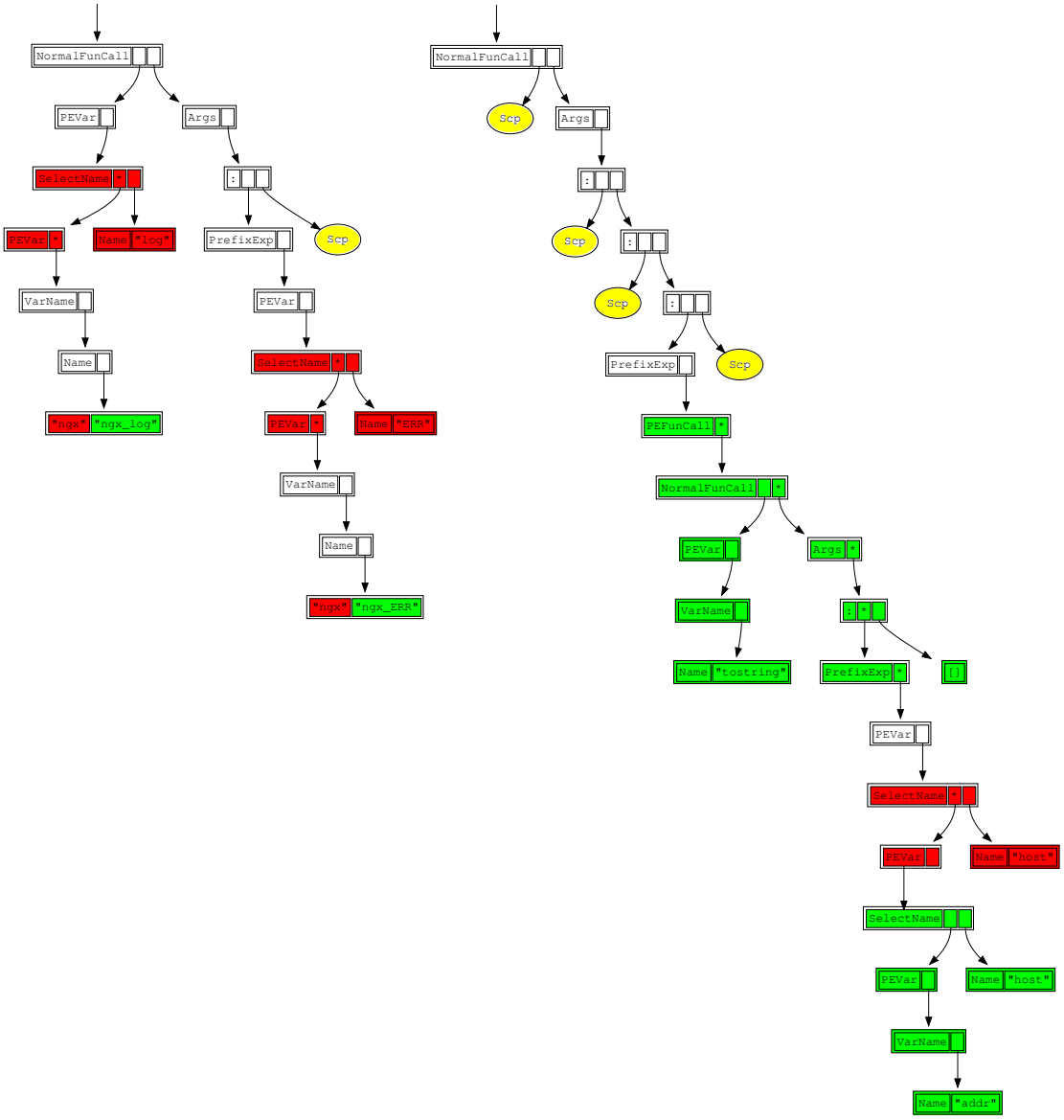


Figure 1.1: Diffs as changes over trees

patches differ, the two patches are trivially disjoint. Looking back at Figure 1.1, at each place where the two patches are different, either the left patch or the right patch is equal to `Scp`. Since the domain of `Scp` matches any subtree, it in particular matches the subtree that the other patch would generate. Hence, we can reconcile any *Scp* with any patch.

Language specific tools for e.g. XML, Java, and C++ have been developed in the past that parse files into abstract syntax trees using their context-free grammars and show promising results [14]. However, there is one big downside to the structured approach. For each language that we want to support, language-specific tooling has to developed [1].

Recently, studies have been conducted to utilise datatype-generic programming to implement efficient structured diffing algorithms in a language-agnostic way [13, 31, 25]. Haskell or Agda datatypes are used to describe language ASTs. They then utilise datatype-generic programming to describe changes between values of these datatypes in a language-agnostic way. However, these approaches all work on a flattened representation of ASTs, which makes it hard to reason about merges. Furthermore, though in theory these algorithms are quadratic, existing implementations are not well-optimised, which makes them not suited for real-world usage. Also, previous approaches can not guarantee that performing a merge produces a tree that is well-formed, or in other words, *well-typed*. Let us illustrate with an example. Consider the following approximation of the Lua AST.

```
data Expr = Val Int
data Stmt
  = If Expr [Stmt]
  | Asg Name Expr
  | Print Expr
```

It would then be invalid if a diff would produce an AST, that would put an assignment in the body of the *If* statement.

```
invalid :: Stmt
invalid = If (Asg "yo" (Val 3)) [Print (Val 3), Print (Val 4)]
              ^^^^^^^^^^^^^^^^^^
            Expected `Expr` but got `Stmt`
```

A novel approach by Miraldo, Dagand, and Swierstra [21] defines patches that follow the same tree-like structure that the source and destiniation value of the patch have. Using this richer structure, it is then possible to reason about merges [20], as we have demonstrated in Figure 1.1. However, their approach lacks an efficent algorithm for generating these patches. Moreover, their patch datatype can only describe changes between *regular* datatypes, but most ASTs need mutually recursive datatypes to be described correctly. For example, the *Stmt* datatype from above is *not* regular, as it not only recurses back into *Stmt*, but also into *Expr*.

1.3 Contributions

In this thesis, we show that type-safe, generic structured diffing is not only theory, but also can be practically used in the real world on actual code repositories, regardless of programming language. More specifically, this thesis makes the following contributions:

- We port the work of Miraldo, Dagand, and Swierstra [21] to Haskell using the `generics-mrsop` library. More importantly, we add support for diffing and merging of mutually recursive datatypes. This allows us to diff and merge a wide variety of programming language ASTs. (Section 4.4)

	Edit scripts [13]	Structured diff [21]	This thesis
Fast diff	In theory	No	Yes
Type-safe diff	Yes	Yes	Yes
Easy merge	No	Yes	Yes
Type-safe merge	Yes	Yes	Yes
Mutual recursion	Yes	No	Yes

Table 1.1: Contributions at a glance

- We implement an optimised version of the *gdiff* algorithm in Haskell. We show that modern Haskell language extensions allow us to reduce the amount of singleton type values to make the patches typecheck, reducing the memory footprint and running time of the algorithm significantly, allowing us to diff large files. (Section 4.3)
- We show that we can implement a quadratic algorithm for generating structured patches by using *gdiff* as an oracle. This allows us to use structured diffs and merges on large ASTs in reasonable time. (Section 4.5)
- We have instantiated our framework for both the Clojure and Lua programming languages and mined merge conflicts from many popular GitHub repositories. We evaluate the performance of our diffing algorithm on this dataset. Furthermore, we evaluate our merging algorithm on this dataset as well. We show, though we use a rather naive merging algorithm, that we are able to solve around 15 percent of conflicts automatically for Clojure, and 13 percent of conflicts for Lua. (Chapter 5)

2
Background - Generic programming

Haskell allows us to straightforwardly represent abstract syntax trees using algebraic datatypes. For example, a language for integer expressions could be defined as:

```
expr                                    data Expr
  ::= expr '+' expr                       = Add Expr Expr
   | 'let' name '=' expr 'in' expr        | Let String Expr Expr
   | int                                  | Val Int
```

Combined with its rich ecosystem for parser-combinator libraries this makes Haskell particularly well-suited for writing tools manipulating programs. It comes to no surprise that Hackage has a rich ecosystem of language ASTs and parsers [1]. For the purpose of building a library for generically diffing and merging programming language ASTs, it is convenient that much of the language-specific work has already been done for us. One question remains; how dowe implement functionality not speficially for one language, but for any programming language AST in general? This is where generic programming libraries come into play. They give us a unified view of the structure datatypes. We then implement desired behaviour over this unified view, and use that to instantiate this behaviour for each datatype for which we desire such functionality. This allows us to implement diffing and merging over ASTs that work for any AST.

In this chapter, we will give an introduction to datatype generic programming. First, we will describe Haskell's built-in support for generic programming. We will describe shortcomings and limitations to this approach of generic programming, and will then present a generic programming library in Agda that is better suited for the purpose of calculating diffs and merges. Using this generic programming library in Agda, we will then present two existing approaches to datatype-generic diffing [21, 13]. in Chapter Both make heavy use of dependent types.

2.1 Datatype-generic programming

Datatype generic programming is an umbrella term for techniques that allow us to define functions in such a way, that they work for many datatypes, instead of repeatedly writing similar functions for different datatypes. One form of generic programming that most Haskell programmers should be familiar with is Haskell's built-in support for deriving certain typeclasses automatically. For example, it allows us to define implementations for pretty-printing or checking datatypes for equality, without having to write those by hand:

```
deriving instance Show Expr
deriving instance Eq Expr
```

[1] `http://hackage.haskell.org/packages/tag/language`

```
> Val 3 ≡ Val 4
False
> show (Add (Val 3) (Val 4))
"Add (Val 3) (Val 4)"
```

However, Haskell's deriving mechanism only works for a predefined collection of typeclasses and functions. It does not allow the user to define new functionality that works generically.

Many libraries have spawned around the concept of generic programming; giving programmers a set of tools to reason about and program with datatypes in a generic fashion, such that end-users can define their own automatically derivable behaviours [12, 26, 16, 32, 33]. Even today it is still an activate area of research [23, 28]. Most generic programming libraries share a similar approach. They define some common representation of datatypes on which algorithms are defined. They then define a mapping between datatypes and their common representation. By composing this mapping with the algorithm defined on the generic representation, we get an implementation for any datatype that has such a representation for free. These generic representations are automatically generated for supported datatypes either through *TemplateHaskell* [29] or through built-in support by the compiler [12, 16]. How this representation is defined, is what differentiates these libraries. As we will see, trade-offs are made between type-safety, expressiveness, efficiency and ease of use when defining such a representation.

The `regular` library serves as the main inspiration for GHC's built-in generic programming library *GHC.Generics*. The `regular` library presents this mapping between representation and type using a typeclass with an associated type family [4], which gives us a *type-safe* mapping between the type and its original representation, as each representation *Rep a* is tagged with the type it is representing.

```
class Generic a where
  type Rep a :: * → *
  from :: a → Rep a a
  to :: Rep a a → a
```

The representation itself consists of a fixed set of *pattern functors* [26]. These are a set of functorial datatypes that allow us to describe the shape of a datatype using sums, products, constants and recursive positions. Algebraic Datatypes lend their name from the fact that datatypes in Haskell consist of *sums and products*. Hence, a generic representation of a datatype will consist of sums and products as well. By taking the *fixed point* of this representation, we get a value that is isomorphic to the value that it represents.

$$Fix\ (Rep\ a) \equiv a$$

```
data U x = U    -- to mark values with no fields (Nothing)
data K i x = K i    -- constant fields in products
data I x = I x    -- recursive positions in products
data (f × g) x = f x × g x    -- products
data (f + g) x = Inl (f x) | Inr (g x)    -- sums
```

Using the Template Haskell rules defined by `regular`, the following *Generic* instance is generated for the *Expr* datatype.

```
data Expr = Val Int | Add Expr Expr
instance Generic Expr where
  type Rep Expr = K Int + (I × I)
```

```
from (Val i) = Inl (K i)
from (Add e1 e2) = Inr (I e1 × I e2)
to (Inl (K i)) = Val i
to (Inr (I e1 × I e2)) = Add e1 e2
```

2.1.1 Writing generic algorithms

In order to demonstrate how to use the `regular` library to do generic programming, we will implement a very simple generic functions that counts the number of nodes in a datatype. First, we define a typeclass that will tell for each pattern functor, how to calculate its size.

```
class GSize (f :: ∗ → ∗) where
  gsize :: (a → Int) → f a → Int
```

U and K are easy. U is an empty product, and trivially has no children, so there are no more nodes to count. Similarly, we do not consider constants as nodes in our tree, but as values of our nodes, hence they do not influence the node count.

```
instance GSize U where
  gsize _ U = 0
instance GSize (K a) where
  gsize _ _ = 0
```

Products and sums are interpreted as expected. If a node has multiple children, we add the sizes of those children together recursively. If there is a choice of constructors, we recurse into the constructor that was specified.

```
instance (GSize f, GSize g) ⇒ GSize (f × g) where
  gsize rec (x × y) = gsize rec x + gsize rec y
instance (GSize f, GSize g) ⇒ GSize (f + g) where
  gsize rec (Inl x) = gsize rec x
  gsize rec (Inr y) = gsize rec y
```

When we find a recursive position I, it means we need to recurse. Hence we call the recursor *rec* to find out the size of the node.

```
instance GSize I where
  gsize rec (I x) = rec x
```

We then complete the definition of our generic algorithm by tying the recursive knot. Each time we hit a recursive position (and thus a node in our tree), we increase the size by 1. We then convert the recursive position into its generic representation using *from* and continue recursing. We set the recursor *rec* to be the *size* function, such that every occurrence of I recursively calls *size* again.

```
size :: (Generic a, GSize (Rep a)) ⇒ a → Int
size = 1 + gsize size (from x)
```

This concludes the implementation of a generic *size* function. And indeed, the implementation works as expected, regardless of what datatype we give it.

```
> size (Val 3)
1
```

```
> size (Add (Val 3) (Val 3))
3
> size (Just 3)
1
> size (1 : 2 : 3 : [ ])
4
```

2.1.2 Deep versus Shallow representations

It is no coincidence that the generic representation of the `regular` library consists solely of *functors*. We can turn any non-recursive functor f into a recursive datatype, by taking the least fixpoint over f [18].

$$\textbf{data}\ Fix\ f = Fix\ \{unFix :: (f\ (Fix\ f))\}$$

The generic deriving mechanism of the `regular` library generates combinations of pattern functors, such that the following isomorphism holds:

$$Fix\ (Rep\ a) \cong a$$

There are all sorts of generic traversals that we can define over just *Fix* [18]. For example, we can define a function for performing generic folds.

$$\begin{aligned} &cata :: Functor\ f \Rightarrow (f\ a \to a) \to Fix\ f \to a \\ &cata = f \circ fmap\ (cata\ f) \circ unFix \end{aligned}$$

This suggests that it would be more useful if our `Generic` typeclass would have the following functions instead:

$$\begin{aligned} &from :: a \to Fix\ (Rep\ a) \\ &to :: Fix\ (Rep\ a) \to a \end{aligned}$$

However, the original formulation is at least as powerful as this one, as we can define the deep representation in terms of the shallow representation:

$$\begin{aligned} &from' :: (Generic\ a, Functor\ (Rep\ a)) \Rightarrow a \to Fix\ (Rep\ a) \\ &from' = Fix \circ fmap\ from' \circ from \\ &to' :: (Generic\ a, Functor\ (Rep\ a)) \Rightarrow Fix\ (Rep\ a) \to a \\ &to' = to \circ fmap\ to' \circ unFix \end{aligned}$$

Given that all our pattern functors are trivially both *Functor* and *Foldable*, we can now straightforwardly define the function *size* above in terms of *cata* and $sum :: f\ a \to Int$.

$$\begin{aligned} &size' :: (Generic\ a, Functor\ (Rep\ a), Foldable\ (Rep\ a), GSize\ (Rep\ a)) \Rightarrow a \to Int \\ &size' = cata\ ((1+) \circ sum) \circ from' \end{aligned}$$

The benefit of using a deep representation is immediately clear. We were able to distill the multi-line implentation of *size* from before to just a single line of code!

When to use a shallow versus a deep representation is a matter of taste. Working on a deep representation involves less boilerplate, as one doesn't have to use typeclasses to drive the recursion. Though on the other hand, converting to a deep representation is costly, as we need to recursively wrap each recursive position into a *Fix*. Luckily, Haskell's lazy semantics make sure that the deep conversion only gets evaluated as far as needed for a generic function to complete. Though when implementing an algorithm like *gsize*, where we need to observe the entire recursive structure anyhow, it can be more performant to opt for the shallow representation instead, at the cost of a more verbose implementation [26].

2.1.3 How to deal with non-regular recursion

The representation using pattern functors unfortunately has more downsides. As the name of the library suggests, it limits us to only being able to represent *regular* types. Regular datatype are datatypes where the right-hand side of the datatype can only directly mention the left-hand side of the datatype, and not recurse into other datatypes[2]. However, Haskell supports way more types than just regular types. Datatypes can be *nested*, meaning that the recursive call to the lest-hand side is inside that of another datatype.

$$\textbf{data}\ \mathit{Rose}\ a = \mathit{Node}\ a\ [\mathit{Rose}\ a]$$

Furthermore, multiple datatype declarations together can form a *family of mutually recursive datatypes*. Many programming language ASTs use both nested and mutually recursive datatypes. As soon as a language has expressions, statements and definitions, support for mutual recursion is often already needed. For example, the small AST below containing just expressions and statements, is a family of mutually recursive datatypes.

$$\textbf{data}\ \mathit{Expr} = \mathit{Lambda}\ \mathit{String}\ \mathit{Stmt}$$
$$\textbf{data}\ \mathit{Stmt} = \mathit{Asg}\ \mathit{String}\ \mathit{Expr} \mid \mathit{Comma}\ \mathit{Stmt}\ \mathit{Stmt} \mid \mathit{Ret}\ \mathit{Expr}$$

Also, more exotic datatypes like *GADTs* and existentials are not regular either. Thus, at a first glance, using pattern functors as our representation seems too limiting for the purpose of defining generic diffing and merging algorithms, as we cannot use it on most non-trivial ASTs.

GHC.Generics, the generic programming library that gets shipped with GHC [16], is a slight variation of the `regular` library that does support a wider range of Haskell datatypes. It 'solves' the problem by not marking recursive positions explicitly in the representation. Instead, it reuses the constant functor K for both constants and recursive positions. This makes the representation less precise, but more expresive. For example, the representation of *Expr* and *Stmt* generated by GHC Generics looks as follows:

$$\textbf{type}\ \mathit{Rep}\ \mathit{Expr} = K\ \mathit{String} \times K\ \mathit{Stmt}$$
$$\textbf{type}\ \mathit{Rep}\ \mathit{Stmt} = (K\ \mathit{String} \times K\ \mathit{Expr}) + (K\ \mathit{Stmt} \times K\ \mathit{Stmt}) + K\ \mathit{Expr}$$

The library does not distinguish between recursive or non-recursive positions at all, side-stepping the problem of recursion completely. Instead, the end-user of the library has to write extra machinery to decide how recursion should happen. By making the generic representation less precise, a wider range of datatypes is supported.

The writer of the generic algorithm will write instances for the pattern functors just like in `regular`. However, the end-user of the generic algorithm will have to provide the algorithm with the set of recursive positions such that the algorithm knows where to recurse. It does this by requiring the end-user to write an instance of a typeclass. To write the *size* function using GHC Generics, we define an auxiliary typeclass *Size* that marks recursive positions, and also change the type of *gsize* slighty, as it doesn't deal with recursive positions itself anymore.

$$\textbf{class}\ \mathit{Size}\ a\ \textbf{where}$$
$$\quad \mathit{size} :: a \to \mathit{Int}$$
$$\textbf{class}\ \mathit{GSize}\ f\ \textbf{where}$$
$$\quad \mathit{gsize} :: f\ a \to \mathit{Int}$$

We then change the instance of *GSize* for the constant functor K to delegate its implementation to whatever is inside K:

```
instance Size a ⇒ GSize (K a) where
  gsize (K x) = size x
```

The rest of the implementation of *GSize* stays exactly the same. The end-user now writes instances for *Size* that drive recursion. Values that have an instance for *Size* will be treated as recursive positions, whilsts values that do not have an instance for *Size* will be treated as constants. By delaying this choice to the end user, instead of forcing recursive positions to be of exactly one type using the identity functor, we can support a larger amount of datatypes.

```
instance Size String where size = 0    -- constant
instance Size Stmt where
  size = gsize ∘ from
instance Size Expr where
  size = gsize ∘ from
```

There is a clear downside to this approach though. By removing the explicit recursive positions in the representation, we can not convert to a deep rerpesentation anymore. This means that the implementation of generic algorithms require more boilerplate. Even worse, if we wants to use such a generic algorithm on a datatype that we define, will need to provide instances to the *Size* typeclass for each datatype in our family of datatypes on which we want to run *size*. The damage can be controlled slightly with the use of *default signatures*. All implementations of *Size* are the same, and hence we can provide the end-user a default implementation.

```
class Size a where
  size :: a → Int
  default size :: (Generic a, GSize (Rep a)) ⇒ a → Int
  size = gsize ∘ from
```

When the end-user then wants to use our library, they do not have to give an explicit implementation of *size* themselves.

```
instance Size String where size = 0    -- constant
instance Size Stmt
instance Size Expr
```

Though it is great that we can now support generic programming on families of mutually recursive datatypes, this approach is not ideal. In Chapter 4, we will show that we can construct a representation of types that both supports mutual recursion and marks recursive positions inside the representation explicitly, giving us both ease of use and expresiveness.

2.2 Datatype-generic programming in Agda

Because most of the previous work on datatype-generic diffing has been formalised in Agda, we will present a generic universe for *regular* datatypes in Agda, such that we can present techniques and algorithms in a uniform way. Though similar to the way regular datatypes are encoded in Haskell, some tweaks were made to make the universe better suited for the purpose of diffing.

A diff that changes an expression to a statement could be seen as function *Expr* → *Stmt*. Such a transformation would then be defined by pattern matching on *Expr*, and moving the required bits of information into some value of *Stmt*. However, in the generic case, we want to express such a transformation as *Rep Expr* → *Rep Stmt*. The problem, however is that *Rep Expr* doesn't tell us

anything about the shape of the generic representation. It only tells us that it is *some* functor $* \to *$, but gives us no clue how this functor is actually structured. Though it is highly likely that GHC is generating nested linked lists of sums and products, it would have been a perfectly valid choice to generate balanced binary trees instead

$$a :+: (b :+: (a :*: (b :*: (c :*: d)))) \sim (a :+: b) :+: ((a :*: b) :*: (c :*: d))$$

In the implementation of *size* , due to the fact that summing numbers is associative and commutative, it will always give the same answer, regardless of how *Rep a* is structured. However, one can imagine that for other algorithms, a sudden change in structure could be problematic. For every algorithm you have to carefully construct typeclass instances for I, K, $:+:$, $:*:$ such that you traverse the generic structure in the correct and expected order.

To tackle this problem, instead of assigning each type a an opaque functor $Rep\ a :: * \to *$ directly, we assign it a $Code\ a :: SOP$ where SOP is a concrete datatype which describes a's sums of products structure that can be inspected explicitly [32].

```
record Generic (A : Set) : Set where
  field
    sop  : A → SOP
    from : A → Rep (sop A)
    to   : Rep (sop A) → A
```

SOP is simply a sum of products of atoms, represented as a double nested list

```
data Atom : Set where
  K : (k : Konst) → Atom
  I : Atom

Prod : Set
Prod = List Atom

SOP : Set
SOP = List Prod
```

All instances of *Generic* thus share the same structure *SOP*. We then define a single mapping *Rep*, that is the same for *all* datatypes, that maps *SOP* to a corresponding functor in a predictible way. The easiest choice for such a mapping would be to simply translate our codes back to sum and product functors.

```
Rep : SOP → Set → Set
Rep [[a, b, c, ...], [d, ...], [e, ...], ...] =
  (K a :*: (K b :*: K c :*: ...)) :+: ((K d) :+: (K e :+: ...))
```

However, that would not be very productive, as we would simply be throwing away all the structured information of our code, to yet again build up an opaque representation. Instead, we define two indexed datatypes that preserve the structure of the codes, and allows us to discover this structure by means of pattern matching. One datatype *NP* for n-ary products, and a datatype *NS* for n-ary sums.

```
data NP (P : k → Set) : List k → Set where
  []  :                        NP p []
  (::) : P x → NP p xs →       NP p (x :: xs)

data NS (P : k → Set) : List k → Set where
  H : P x       →              NS P (x :: xs)
  T : NS P xs →                NS P (x :: xs)
```

```
⟦_⟧P : Prod → Set → Set
⟦ π ⟧P X = NP (λ α → ⟦ α ⟧A X) π

⟦_⟧S : SOP → Set → Set
⟦ σ ⟧S X = NS (λ π → ⟦ π ⟧P X) σ

⟦_⟧A : Atom → Set → Set
⟦ K κ ⟧A X = ⟦ κ ⟧K
⟦ I ⟧A X = X
```

Our function *Rep* is called ⟦_⟧*S* here, which maps codes to their pattern functor, but this time defined in terms of *NS* and *NP*. Atoms are either mapped to the constant functor or to the identity functor to mark recursive positions. Finally, we tie the recursive knot using `Fix`. As its type suggest, given a code σ it gives is back a type (signified by the return type *Set*).

```
data Fix (σ : SOP) : Set where
  ⟨_⟩ : ⟦ σ ⟧S (Fix σ) → Fix σ
```

Not only can we now pattern match on the codes that describe the type of our datatypes, we can also pattern match on their representation in a uniform way because all datatypes share the same type for representations. For example, we can have a function *split*, that pattern matches both on the codes and the representation of a list of fields, in order to split the list of fields into two lists of fields.

```
split : ∀ {a P} {f₂ : List a} (f₁ : List a) → NP P (f₁ ++ f₂) → NP P f₁ × NP P f₂
split []       xs = [] , xs
split (l :: ls) (x :: xs)
  = let xs₀ , xs₁ = split ls xs
    in (x :: xs₀) , xs₁
```

When programming with datatypes we use constructors to construct values of a sum, by providing the constructor its required arguments, and dually pattern match on constructors to get access to its fields. Programming in this way is very natural, and we can introduce this style of programming for our SoP universe. We can represent a constructor as the task of picking a single element from our list of products *SOP*.

```
Constr : SOP → Set
Constr code = Fin (length code)
```

Furthermore, given a *code* : *SOP* and a constructor, we can look up the fields of the constructor.

```
typeOf : (code : SOP) → Constr code → Prod
typeOf []       ()
typeOf (t :: sop) zero    = t
typeOf (t :: sop) (suc n) = typeOf sop n
```

Given a constructor *c* and values for its fields, we can reconstruct a value.

```
inj : {code : SOP} {X : Set} → (C : Constr code) → ⟦ typeOf code C ⟧P X → ⟦ code ⟧S X
inj zero p = H p
inj (suc C) p = H (inj C p)
```

Dually, we can define a *view* [17] , that given some value in our SoP universe, tells us its constructor and its fields.

```
data View {code : SOP} {X : Set} : ⟦ code ⟧S X → Set where
  tag : (C : Constr code) (p : ⟦ typeOf code C ⟧P X) → View (inj C p)
sop : {code : SOP} {X : Set} → (s : ⟦ code ⟧S X) → View s
sop (here p) = tag zero p
sop (there s) with sop s
... | tag C p = tag (suc C) p
```

Using the *sop* to pattern match, and *inj* to construct values, we can program in a style that is very similar to that of programming with normal first class datatypes.

```
CodeTuple CodeMaybe : SOP
CodeTuple = [[K Int, K String, K Char]]
CodeMaybe = [[], [K Int]]
MkTuple : Constr CodeTuple
MkTuple = 0
Just : Constr CodeMaybe
Just = 1
tupleToMaybe : ⟦ sop Tuple ⟧S X → ⟦ sop Maybe ⟧S X
tupleToMaybe tup with sop tup
... | tag MkTuple fields = inj Just (split [K Int] fields)
```

Working with the SoP universe is very similar to working with normal datatypes in Agda. They are not hard to define, and programming using SoP datatypes is similar to programming with normal datatypes. Now that we have a encoding of generics in Agda, we can start looking at existing implementations of generic diffing and merging.

3
Background - Diffing and merging

3.1 Diffs as edit scripts

Lempsink, Leather, and Löh [13] describe an polynomial time algorithm called *gdiff* for performing type-safe diffs between trees. By modifying an algorithm that finds the longest common subsequence of a string, they can construct an algorithm that calculates an edit script (a list of deletions, copies and insertions) between two typed trees [13]. Their work is based on the Maximum Common Embedded Subtree (MCES) problem by Lozano and Valiente [15]. The idea that Lozano and Valiente described is simple. By working on a depth-first pre-order traversal of a tree, we can treat it like a sequence of characters, and thus can calculate the greatest common subsequence on that sequence.

The original MCES algorithm first converts a tree into a sequence before diffing. The *gdiff* algorithm however, keeps track of a *stack* of trees whilst diffing and applying edit scripts. An edit script operation matches a node against the node label, and the arity of the node. This way, we know how many nodes end up on the stack when applying a deletion, and how many nodes get consumed off the stack when applying an insertion. In Figure 3.1 we see how (untyped) edit script operations affect the stack of trees when applying them. First, there is only one element in the stack. Then after deleting the topmost node `there`, three trees end up on the stack. We then insert a node of arity 2, which consumes two items from the stack. And then we do another insertion, consuming the rest of the stack ending up with a single tree again.

3.1.1 Edit scripts

All the following code in this section are parameterised over the code of the datatype that we are diffing:

```
module ES (σμ : SOP) where
```

In our typed versions of this algorithm, we treat constants and constructors as the characters which the edit script can insert, copy or delete. *cof* lets us treat constructors and constants as the same type.

```
cof : Atom → Set
cof I = Constr σμ
cof (K κ) = ⟦ κ ⟧K
```

Because we know the structure of the datatype we are diffing, we do not have to keep track of arity information in each edit operation explicitly. The arity of a node is already known. It is simply the amount of fields the atom has. In the case of a a constructor, we can get the amount of fields through calling the *typeOf*. Constants we treat as constructors with zero arguments.

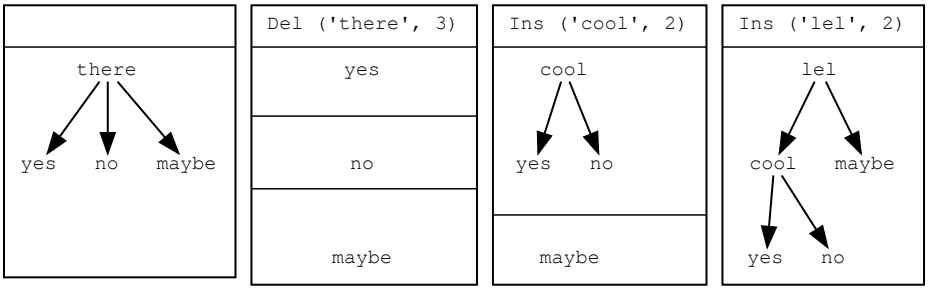


Figure 3.1: Applying an edit script on an untyped tree

$$
\begin{aligned}
&\mathit{fields} : (\alpha : \mathit{Atom}) \to \mathit{cof}\ \alpha \to \mathit{Prod} \\
&\mathit{fields}\ I\ c = \mathit{typeOf}\ \sigma\mu\ c \\
&\mathit{fields}\ (K\ \kappa)\ c = [\,]
\end{aligned}
$$

The edit script is parameterized over its source and target. It keeps track in its types how each operation affects the stack of trees the edit script is going to operate on when applied .

$$
\begin{aligned}
&\textbf{data}\ \mathit{ES} : \mathit{List\ Atom} \to \mathit{List\ Atom} \to \mathit{Set}\ \textbf{where} \\
&\quad \mathit{nil} : \mathit{ES}\ [\,]\ [\,] \\
&\quad \mathit{ins} : \forall\ \{i\ j\ \alpha\}\ (c : \mathit{cof}\ \alpha) \to \mathit{ES}\ i\ (\mathit{fields}\ \alpha\ c +\!\!+ j) \to \mathit{ES}\ i\ (\alpha :: j) \\
&\quad \mathit{del} : \forall\ \{i\ j\ \alpha\}\ (c : \mathit{cof}\ \alpha) \to \mathit{ES}\ (\mathit{fields}\ \alpha\ c +\!\!+ i)\ j \to \mathit{ES}\ (\alpha :: i)\ j \\
&\quad \mathit{cpy} : \forall\ \{i\ j\ \alpha\}\ (c : \mathit{cof}\ \alpha) \to \mathit{ES}\ (\mathit{fields}\ \alpha\ c +\!\!+ i)\ (\mathit{fields}\ \alpha\ c +\!\!+ j) \to \mathit{ES}\ (\alpha :: i)\ (\alpha :: j)
\end{aligned}
$$

In the case of *del*, in order to delete an atom α from the stack, we will match the top of the stack against the constructor c, and if it matches, delete it, and push all the fields of c on top of the stack. Hence the rest of the edit script must at least handle the amount of fields that we just put on top of the stack, signified by $\mathit{fields}\ \alpha\ c +\!\!+ i$. Dually, *ins*ertions require to consume the fields $\mathit{fields}\ \alpha\ c$ of constructor c of atom α in order to produce an α on top of the stack. The neat thing about keeping track of the size of the stack at the type-level, is that this will aid us in enforcing a correct implementation of both *diff* and *apply*.

3.1.2 Applying edit scripts

Because we now treat constants as constructors of zero arguments, we need slight variations of the *inj*, *sop* and *match* functions from the previous section.

$$
\begin{aligned}
&\mathit{inj}' : \{\alpha : \mathit{Atom}\}\ (c : \mathit{cof}\ \alpha) \to [\![\ \mathit{fields}\ \alpha\ c\ ]\!]P\ (\mathit{Fix}\ \sigma\mu) \to [\![\ \alpha\ ]\!]A\ (\mathit{Fix}\ \sigma\mu) \\
&\mathit{match}' : \{\alpha : \mathit{Atom}\}\ (c : \mathit{cof}\ \alpha) \to [\![\ \alpha\ ]\!]A\ (\mathit{Fix}\ \sigma\mu) \to \mathit{Maybe}\ ([\![\ \mathit{fields}\ \alpha\ c\ ]\!]P\ (\mathit{Fix}\ \sigma\mu)) \\
&\mathit{sop}' : \{\alpha : \mathit{Atom}\}\ [\![\ \alpha\ ]\!]A\ (\mathit{Fix}\ \sigma\mu) \to \Sigma\ (\mathit{cof}\ \alpha)\ ([\![\_]\!]A* \circ \mathit{fields}\ \alpha)
\end{aligned}
$$

Instead of working on constructors *Constr s*, they now work on either constructors or constants $c : \mathit{cof}\ \alpha$. And instead of using *typeOf* to get the fields of a constructor, we call *fields* to get the fields of either a constructor or a constant.

Now, let us define a shorthand for a stack of trees, which will make our types slightly easier to read.

$$
\begin{aligned}
&[\![\_]\!]A* \;:\; List\ Atom \to Set \\
&[\![\_]\!]A*\ xs \;=\; [\![\ xs\ ]\!]P\ (Fix\ code)\ \sigma\mu
\end{aligned}
$$

The types of insert and delete look as expected. In order to insert a constructor of atom α, we must have a stack that contains all the fields of that constructor, which we then consume to produce the atom. Dually, deletion pops off an atom α if α's constructor indeed matches the constructor c, and then pushes the remaining fields of c onto the stack. If the constructor of α does not match, deletion fails.

$$
\begin{aligned}
&ins* \;:\; \forall\ \{\alpha\ \alpha s\}\ (c \;:\; cof\ \alpha) \to [\![\ fields\ \alpha\ c\ +\!\!+\ \alpha s\ ]\!]A* \to [\![\ \alpha :: \alpha s\ ]\!]A* \\
&del* \;:\; \forall\ \{\alpha\ \alpha s\}\ (c \;:\; cof\ \alpha) \to [\![\ \alpha :: \alpha s\ ]\!]A* \to Maybe\ [\![\ fields\ \alpha\ c\ +\!\!+\ \alpha s\ ]\!]A*
\end{aligned}
$$

In order to implement insertion, we need to be able to split the stack of *fields* $\alpha\ c\ +\!\!+\ \alpha s$ into two, thus popping of the fields we need to construct the atom. For this, we can use our *split* combinator that we defined earlier.

$$
split \;:\; \forall\ \{a\ P\}\ \{l'\ :\ List\ a\}\ (l \;:\; List\ a) \to NP\ P\ (l\ +\!\!+\ l') \to NP\ P\ l \times NP\ P\ l'
$$

Insertion is then simply defined as popping the fields of the stack, and injecting them together with the constructor, and putting the resulting value back on top of the stack.

$$
\begin{aligned}
&ins* \{\alpha\}\ c\ stack\ = \\
&\quad \textbf{let}\ (f\ ,\ stack')\ =\ split\ (fields\ \alpha\ c)\ stack \\
&\quad \textbf{in}\ inj'\ \{\alpha\}\ c\ f :: stack'
\end{aligned}
$$

For deletion, we need the dual of *split*. Given a set of fields and a stack, we push the fields on top of the stack. Deletion is than just matching on the constructor c, and if there is a match, adding the fields of the constructor to the stack.

$$
\begin{aligned}
&appendNP \;:\; \forall\ \{a\ P\ l\ l'\} \to NP\ P\ l \to NP\ P\ l' \to NP\ P\ (l\ +\!\!+\ l') \\
&appendNP\ [\,]\ \quad xs'\ \quad = xs' \\
&appendNP\ (x :: xs)\ P\ xs'\ = x :: appendNP\ xs\ xs' \\
&del*\ c\ (x :: xs)\ \quad\quad = match'\ c\ x\ \gg\!=\ \lambda\ fields \to just\ (appendNP\ fields\ xs)
\end{aligned}
$$

Now to apply an edit script, we just recurse over the edit script and call *ins*∗ and *del*∗ in the appropriate places.

$$
\begin{aligned}
&applyES \;:\; \forall\ \{txs\ tys\} \to ES\ txs\ tys \to [\![\ txs\ ]\!]A* \to Maybe\ [\![\ tys\ ]\!]A* \\
&applyES\ nil\ \quad\quad [\,]\ = just\ [\,] \\
&applyES\ (ins\ c\ es)\ xs\ = ins*\ c\ \langle\$\rangle\ applyES\ es\ xs \\
&applyES\ (del\ c\ es)\ xs\ = del*\ c\ xs\ \gg\!=\ applyES\ es \\
&applyES\ (cpy\ c\ es)\ xs\ = ins*\ c\ \langle\$\rangle\ (del*\ c\ xs\ \gg\!=\ applyES\ es)
\end{aligned}
$$

3.1.3 Naively generating edit scripts

To calculate the edit script between two sequences, we can naively enumerate all possibilities to go from the source to the target tree, whilst keeping the edit script that is the least 'costly'. Here we define *cost* as the length of the edit script, but one can also think of other cost functions. For

example, one might want to weigh how large the constants are that are being inserted and deleted to find even shorter edit scripts.

```
cost : ∀ {txs tys} → ES txs tys → ℕ
cost nil          = 0
cost (ins c es)   = 1 + cost es
cost (del c es)   = 1 + cost es
cost (cpy c es)   = 1 + cost es
_⊓_ : ∀ {txs tys} → ES txs tys → ES txs tys → ES txs tys
d₁ ⊓ d₂ with cost d₁ ≤? cost d₂
... | yes _ = d₁
... | no  _ = d₂
```

Naively enumerating all options whilst comparing the options by their *cost* yields us an answer for the 'best' edit script.

```
diff : ∀ {xs ys} → ⟦ xs ⟧A* → ⟦ ys ⟧A* → ES xs ys
diff [] []         = ES0
diff [] (y :: ys)  = ins y (diff [] ys)
diff (x :: xs) []  = del x (diff xs [])
diff (x :: xs) (y :: ys) with sop′ x | sop′ y
... | cx , fx | cy , fy | with cx ≟ K cy
... | cx , fx | cy , fy | yes refl = cpy cx (diff (appendNP fx xs) (appendNP fy ys))
... | cx , fx | cy , fy | no _     = ins cy (diff (appendNP fx xs) ys) ⊓
                                     del cx (diff xs (appendNP fy ys))
```

This method is very inefficient. Many sub-calculations are done over and over again. Just like with the greatest common subsequence problem, we can utilise dynamic programming to speed up the algorithm. By caching subresults in a lookup table, we get an algorithm that runs both in polynomial space and time [15]. However, limits to both Agda's interpreter and Agda's Haskell code generator make it hard to implement dynamic programming algorithms, as they do not have a good story for sharing subresults [13]. Hence, we will implement an efficient implementation of this algorithm in Haskell instead. We will go into further detail how to implement this in Section 4.3.

3.1.4 Merging

In order to implement a version control system, we not only need to support diffing between trees, we also need to support merging of such diffs. Merging of patches is needed if two users concurrently worked off some base-version O, but at a later point want to reconcile their work into one patch. Though the work of Lempsink, Leather, and Löh [13] allows us to efficiently calculate diffs, it does not cover the problem of merging. Vassena [31] wrote on the formalisation of a version control system on top of *gdiff*. Two individual edit script operations can be merged into one, if they are not in conflict with eachother. We mean conflicts in the classical sense, which we also know from the classic *diff3* algorithm. in conflict if:

- both perform an insert, but they do not insert the same value
- both perform a copy, but they are not copying the same value
- one performs a copy, and another a deletion, but they do not operate on the same value

Now, to compare two edit scripts and check if they are mergeable, we should pair up each individual operation of one patch with another, and see if they conflict or not. However, simply pairwise merging two edit scripts is usually not the best strategy. Instead, if one can identify which parts of the edit script are operating on the same values, a merge will be more likely to succeed. Vassena [31] define a notion of *alignment*, which inserts no-op edit operations into an edit script, such that as many pairwise operations as possible are not on conflict. Alignment is defined formally as:

- Two edit operations are aligned, if and only if they have the same source value (and type).
- An edit script is aligned, if and only if their edits are pairwise aligned.

As an example, consider the following lists

$$
\begin{aligned}
o &= [1] \\
a &= [0, 1\ ] \\
b &= [1, 2\ ]
\end{aligned}
$$

We can then imagine the following two edit scripts being generated to go from o to a and b respectively:

$$
\begin{aligned}
&es_1\ es_2\ :\ ES\ [\ List\ Int\ ]\ [\ List\ Int\ ] \\
&es_1\ =\ Ins\ \ (:)\ \$\ Ins\ \ 0\ \$\ Cpy\ (:)\ \$\ Cpy\ 1\ \$\ Cpy\ [\,] \\
&es_2\ =\ Cpy\ (:)\ \$\ Cpy\ 1\ \$\ Ins\ \ (:)\ \$\ Ins\ \ 2\ \$\ Cpy\ [\,]
\end{aligned}
$$

These edit scripts are definitely not aligned. es_1's first operation produces a constructor, whilst es_2 needs to consume a constructor, and hence the two patches are not aligned. However, if we introduce a new edit operation, which trivially always operates on any source and destination value

$$Nop\ :\ ES\ xs\ ys\ \rightarrow\ ES\ xs\ ys$$

then we can re-introduce alignment by padding the two edit scripts with this constructor "no operation" edit:

$$
\begin{aligned}
&es_1\ =\ Ins\ (:)\ \$\ Ins\ 0\ \$\ Cpy\ (:)\ \$\ Cpy\ 1\ \$\ Nop\ \ \ \$\ Nop\ \ \$\ Cpy\ [\,] \\
&es_2\ =\ Nop\ \ \ \$\ Nop\ \ \$\ Cpy\ (:)\ \$\ Cpy\ 1\ \$\ Ins\ (:)\ \$\ Ins\ 2\ \$\ Cpy\ [\,]
\end{aligned}
$$

Now, the edit script is aligned. Furthermore, each pair of edit operations are not in conflict, and hence the edit script can be succesfully be reconciled.

Vassena [31] claim that two edit scripts from the same origin can *always* be aligned by extending the edit scripts with *Nop*s. Alignment is also fundamental in the original `diff3` algorithm. It is what finds relevant parts in two patches that need merging, and what parts are trivially copyable. `diff3` comes with all kinds of heuristics to figure out which parts of the file to actually compare on a line-by-line basis. However, for *gdiff* the authors do not define a strategy for finding good alignments for two edit scripts, but instead leave this as an unanswered problem. Non-deterministically enumerating all alignments would be an option, but this would throw away any desired performance characteristics of our *gdiff* algorithm.

Even if we assume that we can come up with an efficient algorithm for aligning edit scripts, there is still one fundamental problem with this approach. Two edit scripts that are aligned, and also not in conflict, can create a merged patch that is not *well-typed*. In the sense that applying the patch to a value of a datatype does not guarentee that the outcome of the merge patch produces a well-formed value of a datatype. This is problematic. Consider the following *aligned* patches over the *List Int* datatype.

$e_1\ e_2\ :\ ES\ []\ [List\ Int]$
$e_1\ =\ [\ Ins\ (::)\ ,\ Nop,\ \ Ins\ 1\ ,\ Ins\ []]$
$e_2\ =\ [\ Nop,\ ,\ \ \ Ins\ [],\ Nop,\ \ Nop\ ]$

Then we can merge the two, as each consecutive edit operation is not in conflict with eachother. Resulting in the following edit script:

$$merge\ e_1\ e_2\ \equiv\ [Ins\ (::)\ ,\ Ins\ [],\ Ins\ 1\ ,\ Ins\ []]$$

However, this edit-script is not well-typed. If we apply it on e_1, the value we get back is not a *List Int*. The constructor (::) does not take a list as its first argument, but an *Int* However, as its first argument it is passed the empty list `[]`. Hence, applying the resulting patch will fail as it does not produce a well-typed value.

If we try to come up with a general type for merging edit scripts, it is hard to do so:

$$merge\ :\ ES\ xs\ ys\ \rightarrow\ ES\ xs\ zs\ \rightarrow\ ES\ xs\ ?$$

Even though the two edit scripts are aligned, we do not know whether the merge will pick *ys*, *zs*, or a combination of both to create the resulting edit script. It is thus impossible to decide whether the resulting edit script actually produces one of the two, or any at all. This is an unfortunate consequence of the fact that the edit script works on the pre-order traversal of trees, throwing away the structured information of the shapes of the source and target tree.

3.2 Structured Diffs

Edit scripts are efficient to compute, but as it turns out are tricky to merge. The root cause of this is that edit scripts do not preserve enough information about the structure of the trees they were calculated from and hence it is hard to find out what parts of the edit script should be reconciled. And even when we find these parts, we can not guarantee that the resulting patch still produces well-formed trees. Miraldo, Dagand, and Swierstra [21] try to come up with a solution to this problem. Instead of taking an efficient algorithm for diffs like MCES, and finding out how to encode this in a datatype-generic way, and then hope to find a way to merge such a patch, they take the opposite approach. What if instead we define a patch datatype, from which we know it preserves enough structure about the trees it is patching, such that merging is easy. They accomplish this, by capturing a notion of *greatest common coproduct structure* of two values that are being diffed.

3.2.1 Spine

The first part of the patch takes care of the sums in our sums-of-products view of datatypes. The authors describe the *spine* of a patch, which is the largest common coproduct structure between two pieces of data that are being diffed. Three different cases are distinguished in the spine that compares values `x` and `y`. A Spine is parameterised over a type *At* that tells us how to patch atoms, and a type *Al* that tells us how to align to sets of fields.

$$\textbf{data}\ S\ (At\ :\ Atom\ \rightarrow\ Set)\ (Al\ :\ Prod\ \rightarrow\ Prod\ \rightarrow\ Set)\ (s\ :\ SOP)\ :\ Set\ \textbf{where}$$

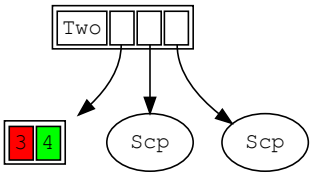
We will now define each constructor of the spine step by step. First of all, the case where the sums `x` and `y` are equal. We simply copy over the content without looking inside either value

$$Scp : S\ At\ Al\ s$$

Next, we have the case where x, and y have the same constructor c, but some of the fields of x and y might be distinct.

$$Scns\ :\ (c\ :\ Constr\ s)\ \rightarrow\ NP\ At\ (typeOf\ s\ c)\ \rightarrow\ S\ At\ Al\ s$$

In that case, we will traverse the fields of x and y pairwise, and patch any field that differs using the type of patches of atoms At. Note that the definition of $Scns$ closely resembles the definition of $[\![\ \pi\ ]\!]P$ in Section 2.2. Signifying that indeed our patch datatype follows the same tree-like structure as our values. Graphically, an $SCns$ looks as follows:



If the two trees do not have matching constructors, then we have to figure out how we change the fields of the first constructor to the fields of the second constructor using Al. If we can indeed change the fields of x into the fields of y then we can change the tree x into the tree y

$$Schg\ :\ (c1\ c2\ :\ Constr\ s)\ \rightarrow\ \{c1\ !=\ c2\}\ \rightarrow\ Al\ (typeOf\ s\ c1)\ (typeOf\ s\ c2)\ \rightarrow\ S\ At\ Al\ s$$

Now given a spine and a value of our SOP representation, we can apply the patch to obtain a new sum value. Lets assume for now we already know how to deal with atoms and alignments through two existing functions *applyAt* and *applyAl* Then we can define applying a spine to a sum.

$$\begin{aligned}
&applyAt\ :\ \forall\ \{\alpha\} \rightarrow At\ \alpha \rightarrow [\![\ \alpha\ ]\!]A\ X \rightarrow Maybe\ ([\![\ \alpha\ ]\!]A\ X)\\
&applyAl\ :\ \forall\ \{\pi_1\ \pi_2\} \rightarrow Al\ \pi_1\ \pi_2 \rightarrow [\![\ \pi_1\ ]\!]P\ X \rightarrow Maybe\ ([\![\ \pi_2\ ]\!]P\ X)\\
&applyS\ :\ S\ At\ Al\ sop \rightarrow [\![\ sop\ ]\!]S\ X \rightarrow Maybe\ ([\![\ \sigma sop]\!]S\ X)
\end{aligned}$$

To apply a Cpy we simply copy over the left hand side.

$$applyS\ Scp\ s\ =\ just\ s$$

To apply a change in constructor, we first check whether the sum we are applying the patch on has the same constructor as the source constructor of the $Schg$ patch, and if its the case we replace the constructor with the target constructor. Then, we convert the old fields to the fields of the new constructor using *applyAl*

$$\begin{aligned}
&applyS\ (Schg\ C_1\ C_2\ p)\ s\ \textbf{with}\ sop\ s\\
&...\ |\ tag\ C_3\ p_3\ \textbf{with}\ C_1 \stackrel{?}{=} F\ C_3\\
&...\ |\ yes\ refl\ =\ inj\ C_2\ \langle\$\rangle\ applyAl\ p\ p_3\\
&...\ |\ no\ \neg p\ =\ nothing
\end{aligned}$$

If we are not changing the constructor, but only the fields, we check whether the patch's constructor and the sum we're patching match, and if so, change the fields pairwise using At.

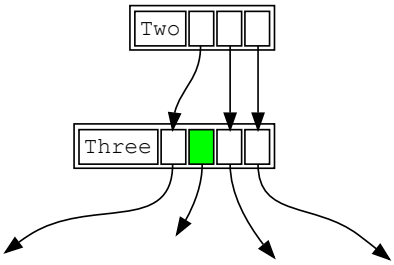
$$\begin{aligned}
&applyS\ (Scns\ C_1\ p_1)\ s\ \textbf{with}\ sop\ s\\
&...\ |\ tag\ C_2\ p_2\ \textbf{with}\ C_1 \stackrel{?}{=} F\ C_2\\
&...\ |\ no\ \neg p\ =\ nothing\\
&...\ |\ yes\ refl\ =\ inj\ C_1\ \langle\$\rangle\ mapNPM\ applyAt\ (zipNP\ p_1\ p_2)
\end{aligned}$$

3.2.2 Alignment

To align two products, we define a type *Al* that looks very similar to the type of edit scripts of *gdiff*. A very important distinction is that *gdiff* deals with constructors and products at the same time, allowing the source and target stacks grow and shrink as we traverse the patch, whilst *Al* only strictly deals with insertions and deletions of entire subtrees. It is really just converting one set of fields into another set of fields, and nothing more. To match two constructors that have different arity, we can delete fields from the source constructor using *ADel* or insert fields at the destination constructor using *AIns ys*.

```
data Al : (At : Atom → Set) : Prod → Prod → Set were
  A0   :                                          Al At []       []
  AX   : ∀ {a xs ys} → At a    → Al At xs ys → Al At (a :: xs) (a :: ys)
  ADel : ∀ {a xs ys} → ⟦ a ⟧A → Al At xs ys → Al At (a :: xs) ys
  AIns : ∀ {a xs ys} → ⟦ a ⟧A → Al At xs ys → Al At p2       (a :: ys)
```

Here below, we see an example of such an alignment, where a value with 3 fields is aligned to a value with *4* fields, by doing an extra *AIns* at the appropriate place, to align the fields of the two constructors.



Applying an alignment to a product is easy. In case of an *AX* we need to transform an atom, which we can do with *applyAt*. When we insert an atom with *AIns* we insert it in the resulting list, and recursively continue applying the alignment. Dually, when we are deleting an atom, we remove it in the resulting list, and continue applying the alignment recursively.

```
applyAl : ∀ {π1 π2} → Al At π1 π2 → ⟦ π1 ⟧P X → Maybe (⟦ π2 ⟧P X)
applyAl (AX at al) (a :: as)  = _ :: _ ⟨$⟩ applyAt at a ⟨*⟩ applyAl applyAt al as
applyAl (Ains a' al) p        = (a' :: _) ⟨$⟩ applyAl applyAl al p
applyAl (Adel a' al) (a :: as) = guard a' ≡ a *⟩ applyAl applyAl al as
```

3.2.3 Atoms

Assuming for now how we are going to tie the recursive knot by means of some *PatchRec* , we can define patches for Atoms. An atom can either point to a recursive position, or to an opaque type. Because we know nothing about the structure of opaque types, the best we can do is describe the

source and destination value And for a recursive position, we use our assumed *PatchRec* to further describe the patch.

```
data At (PatchRec : Set) : Atom → Set where
  set : ⟦ k ⟧K → ⟦ k ⟧K → At PatchRec (K k)
  fix : PatchRec → At PatchRec I
```

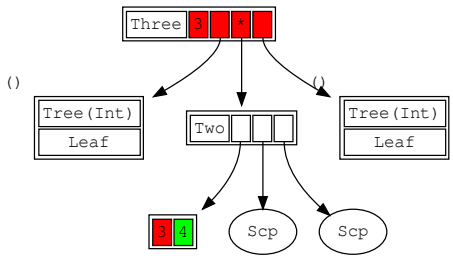
3.2.4 Tying the recursive knot

Now we will handle recursive positions. At the recursive level, three things can happen.

- We can have a change of constructor, which is handled by our spine datatype *S*.
- We can peel off a constructor and its fields, exposing a new recursive position
- We can wrap a recursive position with a constructor and extra fields

```
data Alμ : Set where
  spn : {s : SOP} → S (At Alμ) (Al (At Alμ)) s → Alμ
  ins : {s : SOP} → (c : Constr s) → Ctx (typeOf s c) → Alμ
  del : {s : SOP} → (c : Constr s) → Ctx (typeOf s c) → Alμ
```

Ctx is a list of fields ⟦ *p* ⟧*P* but with a hole in it pointing to an *Alμ* that tells how that specific field should be patched further. It resembles much the idea of a Zipper. Graphically, a context looks like this:



In code, they are defined as follows. We either are at the hole (at a recursive position *I*) and have a patch for that hole followed by the rest of the *NP*, or we have a single element of the *NP* followed by a context which will eventually contain the hole. This way we have an *NP* with exactly one hole in it.

```
data Ctx : Prod → Set where
  here : ∀ {π} → (spμ : Alμ) (atμs : NP (λ α → ⟦ α ⟧A (Fix μσ)) π) → Ctx (I :: π)
  there : ∀ {α π} → (atμ : ⟦ α ⟧A (Fix μσ)) (alμs : Ctx π) → Ctx (α :: π)
```

To apply a deletion context, we match the constructor of the context against the constructor of the value we're patching. If they match, we delete the constructor and all the fields *except* the recursive position that is pointed at by the context, and then we recursively apply our *Almu* patch on that

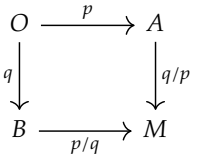


Figure 3.2: Incorperating the changes of p into q and applying it gives the same result as incoorperating q into p

exposed value. In a sense, we are peeling off a layer of recursion. Dually, an insertion context adds an extra layer of recursion. Note that for insertion, we do not have to match its constructor, as we are not removing things, but simply wrapping an extra layer around the existing datatype.

$$
\begin{aligned}
&\mathit{applyAl}\mu \;:\; \mathit{Al}\mu \to \mathit{Fix}\;\mu\sigma \to \mathit{Maybe}\;(\mathit{Fix}\;\mu\sigma)\\
&\mathit{applyAl}\mu\;(\mathit{del}\;C\;\mathit{al}\mu)\;x \;=\; (\mathit{matchCtx}\;\mathit{al}\mu \bullet \mathit{match}\;C)\;(\mathit{unfix}\;x)\\
&\mathit{matchCtx}\;(\mathit{here}\;\mathit{sp}\mu\;\mathit{at}\mu s)\;(x :: p) \;=\; \mathit{applyAl}\mu\;\mathit{sp}\mu\;x\\
&\mathit{matchCtx}\;(\mathit{there}\;\{\alpha\}\;\mathit{at}\mu\;\mathit{al})\;(\mathit{at} :: p) \;=\; \mathit{matchCtx}\;\mathit{al}\;p
\end{aligned}
$$

With this, we have a patch datatype that closely follows the structure of the two trees it is converting between. As we will see in the next section, this structure will make merging quite a bit easier.

$$
\begin{aligned}
&\mathit{Patch} \;:\; \mathit{SOP} \;\to\; \mathit{Set}\\
&\mathit{Patch}\;s \;=\; S\;\mathit{Atmu}\;(\mathit{Al}\;(\mathit{At}\;\mathit{Al}\mu))\;s
\end{aligned}
$$

3.2.5 Merging patches

The promise of structured patches is that they will provide a better story for merging. Let us recall the idea behind a three-way merge [19]. When two persons edit the same tree O, one will end up with tree A through patch p and the other will end up with tree B through patch q.

$$B \xleftarrow{q} O \xrightarrow{p} A$$

Now, we want to produce a tree M that contains the changes made in both A and B. To do so, we calculate a new patch p/q, that has all the changes done by p, adapted to the changes that q makes. Such an operation is similar to *applyAl*μ, which applies a patch to a tree. But now instead, we apply the patch to the tree structue of another patch. However, it is important that this patch adapatation function commutes for it to be a well-behaved merge. That is, we can either adapt p to q or q to p, and we will still end up with the same value M if we apply them to A or B [25]. This is sometimes called the *residual* of p and q [10, 22]. The commutative diagram illustrating this merge property is shown in Figure 3.2.

Looking back at the example from the introduction, we noted that both patches were *disjoint*. At all places where p and q were different, p would not restrict the domain of q, and q would not restrict the domain of p. If two edits are disjoint, we can trivially *apply* the patch p to the patch q.

Miraldo [20] provides a Agda proof that if two patches p and q are disjoint, then p will indeed apply to q.

$$\mathit{merge} \;:\; \{\mathit{disj} \;:\; \mathit{disjoint}\;p\;q\} \;\to\; (p \;:\; \mathit{Al}\mu) \;\to\; (q \;:\; \mathit{Al}\mu) \;\to\; \mathit{Al}\mu$$

Furthermore, they provide an Agda proof that this simple implementation of just applying the changes of patch p to patch q is indeed commutative as in Figure 3.2 when the patches p and q are disjoint. So, an effective and correct strategy for merging structured patches is:

- First, check if p and q are disjoint
- Second, either adapt p to q or q to p, depending on whether you want to go from A to M or from B to M

We will now briefly sketch the criterum of disjointness.

$$disjoint : Al\mu \to Al\mu \to Set$$

First of all. A patch is never disjoint from itself.

$$\begin{aligned} &disjoint\ (del\ C_1\ s_1)\ (del\ C_2\ s_2) &&= \bot \\ &disjoint\ (ins\ C_1\ s_1)\ (ins\ C_2\ s_2) &&= \bot \end{aligned}$$

An insertion context can be applied to any tree x. It does not restrict the domain of a patch in any way. Hence, if p is indeed an insertion, then we can adapt q to incoorporate the insertion that p performed, as long as the patch to which the insertion context of p points it disjoint from q.

$$\begin{aligned} &disjoint\ (ins\ C_1\ s_1)\ x &&= disjoint\ (getCtx\ s_1)\ x \\ &disjoint\ x\ (ins\ C_1\ s_1) &&= disjoint\ (getCtx\ s_1)\ x \end{aligned}$$

Deletions do restrict the domain of a patch. However, there are still two cases where a deletion can be disjoint from another patch. A deletion and an *Scp* node are disjoint. This is because *Scp* is a patch that matches any size tree. If you 'delete' something from an *Scp*, you will still end up with an *Scp*. Not only is a deletion disjoint with an *Scp*, anything is trivially disjoint from an *Scp*

$$\begin{aligned} &disjoint\ x\ (spn\ Scp) &&= \top \\ &disjoint\ (spn\ Scp)\ x &&= \top \end{aligned}$$

Finally, a deletion might be disjoint with a spine, if it happens to be that they operate on the same constructor

$$\begin{aligned} &disjoint\ (del\ C_1\ s_1)\ (spn\ (Scns\ C_2\ at_2)) &&= \Sigma\ (C_1 \equiv C_2)\ (\lambda\ \{refl \to disjCtxAt\ s_1\ at_2\}) \\ &disjoint\ (spn\ (Scns\ C_2\ at_2))\ (del\ C_1\ s_1) &&= \Sigma\ (C_1 \equiv C_2)\ (\lambda\ \{refl \to disjAtCtx\ at_2\ s_1\}) \end{aligned}$$

In all other cases, a deletion is not disjoint.

$$\begin{aligned} &disjoint\ (del\ C_1\ s_1)\ x &&= \bot \\ &disjoint\ x\ (del\ C_1\ s_1) &&= \bot \end{aligned}$$

Spines *can* be disjoint. We already saw that in particular, an *Scp* is always disjoint with another patch. Moreover. two spines can be disjoint, if their chilrden are pairwise disjoint. If one spine is an $Scns\ C_1\ at$ and the other spine is an $SChg\ C_3\ C_4\ al$, then they will only be disjoint if it happens to be that $C_1 \equiv C_3$ and *at* and *al* are recursively disjoint. If however both patches change the constructor they are not disjoint.

$$disjoint\ (spn\ (Schg\ C_1\ C_2\ al_1))\ (spn\ (Schg\ C_3\ C_4\ al_2)) = \bot$$

We have only sketched the implementation of *disjoint* but have so far omitted the implementation of *merge* itself. The implementation of *merge* will be very similar to that of $applyAl\mu$, but instead operates on patches instead of trees. A full implementation can be found in Chapter 4.

Discussion

One problem that remains is how we would generate such patches. Miraldo, Dagand, and Swierstra [21] do not define an efficient algorithm for generating patches. Instead, they define a non-deterministic enumeration that enumerates all patches between two trees a and b. Garufi [9] show that by doing a bounded search, using domain specific pruning rules, it is feasible to use non-deterministic generation to find suitable patches. However, this is at a loss of generality, as specific rules had to be created for a specific language . And even then, performance was still not great [9]. Furthermore, structured diffs as presented here only work for *regular* types. However, many programming language ASTs require mutually recursive types to be correctly described. Limiting ourselves to just regular types it too restrictive to use these diffs and merges in the real worlds. In Chapter 4 we will present an efficient algorithm for generating structured patches, that does not require the user to provide domain-specific information for the language that they want to perform diff on, giving us truly best of both worlds. Efficient like *gdiff*, but also the possibility to easily merge patches. Furthermore, the type of patches will be extended to support families of mutually recursive datatypes, allowing us to perform structured diffs on a wide variety of programming languages.

4 Haskell Implementation

Motivation

Agda is a great tool for reasoning about code. However, its code-generation backend is still very experimental. The GHC compiler, on the other hand, is an industry-strength compiler with a great amount of optimization passes. Furthermore, Haskell is lazily evaluated and has a notion of sharing, which is going to be key in making our implementation of diff fast. Recently, GHC has added many language extensions to the language that bring its type system closer and closer to that of Agda [34]. In the future, GHC will even provide a fully dependently typed dialect of the Haskell language [6, 4]. However, as we will see, with the language extensions with which GHC ships today, we can already port much of the Agda code from Section 2.2.

Also as already mentioned in Chapter 2, Haskell comes with a rich ecosystem of community-provided packages through the Hackage package repository. Which would make a generic diffing and merging tool written in Haskell applicable to hundreds of languages without much effort. This motivates us to implement our generic diffing and merging tool in Haskell.

We will often abbreviate parts of code, omit certain type paramters, or only show the types of certain functions to ease readability. A full working copy of the source code can be found at [27].

4.1 A generic representation for Abstract Syntax Trees

Writing algorithms in the style *GHC.Generics* is quite tedious. In Section 2.2 we defined a library for generic types based on structured codes.This style of generic programming is particularly well suited for the purpose of diffing, because not only the traversing but also the manipulation of datatypes is easy. However, that generic universe limits us to just *regular* types. In order to diff and merge programming language ASTs, we need a generic programming library that at least supports families of mutually recursive datatypes, whilst retaining the ergonomics of the structured SoP universe. In this section, we present an implementation of this generic universe in Haskell. The library we will present is `generics-mrsop` [23]. Furthermore, we will show further improvements that we have contributed to this library, and discuss remaining limitations.

4.1.1 Inspectible structure of representation

The `generics-sop` library by de Vries and Löh [32] implements a generic universe very similar to that of the Agda universe presented in Section 2.2. It modifies the *Generic* typeclass by not letting it determine the opaque representation *Rep a* directly, but instead it assigns each type a a structured description of the type $Code_{sop}\ a :: [[*]]$. Each code is then interpreted as a concrete type by the mapping $Rep_{sop}\ (a :: Code_{sop}\ a) :: *$ [32]. The kind of codes is slightly different than the type for

codes in Agda. This is because `generics-sop` does not mark recursive positions explicitly. It does this such that it can support a richer universe than just *regular* types. This is the same trick that *GHC.Generics* employs to support more datatypes.

```
type Rep_sop code = ...
class Generic a where
  type Code_sop a :: [[*]]
  from_sop :: Rep_sop (Code_sop a) → a
  to_sop   :: a → Rep_sop (Code_sop a)
```

Using *DataKinds* an *PolyKinds* [34], we can define the datatypes *NP* and *NS* similarly as we did in Agda. By pattern matching on values of *NS* and *NP*, we learn about the structure of the code. This allows us to shrink, increase, and manipulate codes, by manipulating the underlying values of the representation.

```
data NP (p :: k → *) :: [k] → * where
  NP0 ::                    NP p []
  (:*) :: p x → NP p xs → NP p (x : xs)

elimNP :: (∀x.f x → a) → NP f ks → [a]
elimNP f NP0 = []
elimNP f (k :* ks) = f k : elimNP f ks
```

```
data NS (p :: k → *) :: [k] → * where
  H :: p x      → NS p (x : xs)
  T :: NS p xs → NP p (x : xs)

elimNS :: (∀x.f x → a) → NS f ks → a
elimNS f (H p) = f p
elimNS f (T p) = elimNS f p
```

The representation of a code is then simply the composition of *NS* and *NP*. Because we do not mark recursive positions explicitly, we can interpret our atoms by using the identity functor *I*. In Figure 4.1 we see an instance of *Generic* for a family of datatypes. Of course, `generics-sop` supports automatically deriving these instances using *TemplateHaskell*.

```
newtype SOP (p :: k → *) (code :: [[*]]) = SOP (NS (NP p) code)
type Rep_sop a = SOP I (Code a)
```

de Vries and Löh [32] define many combinators over $Rep_{sop}\ a$, that allow the user to map over, transform, shrink and grow representations. Above, we have defined two useful eliminators *elimNS* and *elimNP*, that allow us to consume a representation all at once to some summary value. Using these combinators, we can define generic functions with ease. Let us revisit our generic *size* function which we introduced in Chapter 2.

```
gsize :: (Generic a, All2 Size (Code a)) ⇒ a → Int
gsize = sum ∘ elimNS ∘ elimNPC (Proxy :: Proxy Size) (mapIK size) ∘ from

instance Size Name where size = 0
instance Size Int where size = 0
instance Size a ⇒ Size [a]
instance Size Func
instance Size Stmt
instance Size Expr
```

Using the combinators *elimNS* and *elimNP* we can define the *gsize* function very tersely. The implementation of *gsize* has little to no boilerplate, except for the explicit marking of recursive positions through the *Size* typeclass.

```
type Name = String
type Args = [Name]
data Func = Func Name Args [Stmt]
data Stmt = Set Name Expr | Call Name [Expr]
data Expr = Var Name | Int Int | Add Expr Expr
instance Generic Expr where
  type Code Expr =  [[Name], [Int], [Expr, Expr]]
  from :: Expr → Rep [[Name], [Int], [Expr, Expr]]
  from (Var n) = SOP $ H $ I n :* NP0
  from (IntA i) = SOP $ T $ H $ I n :* NP0
  from (Add e1 e2) = SOP $ T $ H $ H $ I e1 :* I e2 :* NP0
  to :: Rep (Code Expr) → Expr
  to =  {-analogous -}
instance Generic Func where  {-analogous -}
instance Generic Stmt where  {-analogous -}
```

Figure 4.1: An example of the `generics-sop` type universe

Figure 4.2: generic size function using `generics-sop`. If we want to use *gsize* on a type, we need to write quite some boilerplate.

4.1.2 Explicit recursion for mutually recursive families of datatypes

Though `generics-sop` allows us to inspect and calculate with codes, it only allows us to observe one layer of recursion at a time. Though the boilerplate of inspecting the SOP structure using typeclasses is now gone, we still need a typeclass to explicitly tell where recursion takes place. Doing this for a few types, as in Figure 4.1 is still manageable, but most programming languages have abstract syntax trees that are orders of magnitude larger than this one. That adds up to quite a bit of boilerplate, considering we need to define a typeclass for *each* generic function that we implement, and also then for each generic function have to write an instance marking for *each* recursive position of each datatype in our abstract syntax tree.

As we already saw in Section 2.1, we can explicitly mark recursive positions in the representation of a datatype to remove the need of the *Size* typeclass. So, lets extend our language of codes to allow for explicit recursive positions, analogously to the Agda code in Section 2.2.

```
data Atom (kon :: κ → ∗) = K kon | I
class Generic a where
  type Code_sop a :: [[Atom kon]]
```

This definition will however limit ourselves to just *regular* datatypes. But, *Expr* is not a regular datatype. It is a member of a family of mutually recursive datatypes. If we want to use *gsize* for *Expr*, *Stmt* and friends, we need to handle mutual recursion as well. To do so, we make several to changes. First of all, we do not assign a type a code, but assign a family of types a family of codes.

```
type family Code_mrsop (fam :: [∗]) :: [[[Atom kon]]]
```

Second of all, recursive positions must now explicitly mention into *which* type of the family they recurse, by means of an index into the family of types.

```
data Atom (kon :: κ → ∗) = K kon | I Nat
```

Mapping indexes to types can then be accomplished with a type family that does a lookup into type-level lists:

```
type family Lkup (xs :: [a]) (n :: Nat) :: a where
```

At first glance, the type of atoms seems to be too general. A recursive position can refer to *any* number here, whilst we only want it to be able to refer to numbers n such that $n < Lengthfam$. This could for example be accomplished with a finite type *Fin n* that has exactly n inhabitants, but this would mean we would have to carry around the size of the family around everywhere, which becomes quite unwieldy [23]. Luckily, this turns out not be a problem in practice. If we were to construct an atom that has an index that is out of bounds for *fam* :: [∗], *Lkup* will fail with a type error at *compile time*. Furthermore, end users will never directly use *Lkup* themselves, but is called by the functions *to* and *from* directly. We can even customise the type error to be useful to the end user using GHC's support for custom type errors.

With these changes, The family of codes for *Expr* and friends will look as follows:

```
data Konstant = KName | KArgs | KInt
type Code_mrsop (fam :: [∗]) :: [[Atom (Length fam) kon]]
type family instance Codes_mrsop [Func, [Stmt], Stmt, Expr, [Expr]] =
  [[[K KName, K KArgs, I 1]]          -- Func
  ,[[], [I 2, I 1]]                   -- [Stmt]
  ,[[K KName, I 3], [K KName, I 4]]   -- Stmt
  ,[[K KName], [K KInt], [I 3, I 3]]  -- Expr
  ,[[], [I 3, I 4]]                   -- [Expr]
  ]
```

Of course, if our codes change, we will also need to adjust the representation of codes. First of all, we need to map Atoms to types. We do this with the datatype *NA*, that can interpret an *Atom* to a type, given we tell it how to map constants to types through κ, and how to map recursive positions to types through φ.

```
data NA (κ :: kon → ∗) (φ :: Nat → ∗) :: Atom kon → ∗where
  NA_I :: (IsNat n) ⇒ φ n → NA κ φ (I n)
  NA_K :: κ k → NA κ φ (K k)
```

We define an eliminator for *NA* similar to the eliminators for *NS* and *NP*. if we can map both $\varphi\ n$ and $\kappa\ k$ to a value b, then we can collapse an *NA* into a value b:

```
elimNA :: (∀k.κ k → b) → (∀k.IsNat k ⇒ φ k → b) → NA κ φ a → b
```

Then, our representation of codes is a very slight tweak of Rep_{sop}, such that it is also parameterized over κ and φ.

```
newtype Rep κ φ code = Rep {unRep :: NS (NP (NA κ φ)) code}
```

To eliminate a *Rep* to a single value, we combine *elimNS*, *elimNP* and *elimNA*.

```
elimRep :: (∀k.κ k → a) → (∀k.IsNat k ⇒ f k → a) → ([a] → b) → Rep κ f c → b
elimRep kp fp cat = elimNS (cat ∘ elimNP (elimNA kp fp)) ∘ unRep
```

We want to set $\varphi = Lkup\ fam$, such that *NA_I* selects the correct type. However, this is not possible because *Lkup* is a type family, and type families always need to be fully saturated in GHC. That means that *Lkup* can not appear on its own, but must always have all its arguments applied. The type system currently can not cope with partially applied type families. This might change in the future however, with the introduction of Dependent Haskell [7]. To work around this issue, we define a wrapper datatype around *Lkup* because datatypes *can* appear unsaturated.

```
data El :: [*] → Nat → *where
  El :: IsNat ix ⇒ Lkup ix fam → El fam ix
```

We also define a type family $Idx :: * \to [*] \to Nat$, which tells us the index of an element in a list. Now, we can define our *to* and *from* functions, that convert a given type *ty* into its corresponding *Rep*.

```
class Family κ fam codes | fam → κ codes, κ codes → fam where
  from :: (ix~Idx ty fam, Lkup ix fam~ty) ⇒ ty → Rep kon (El fam) (Lkup codes ix)
  to   :: (ix~Idx ty fam, Lkup ix fam~ty) ⇒ Rep kon (El fam) (Lkup codes ix) → ty
```

Note that we pass *El fam* as the interpretation of recursive positions to *NA*, but we do not pass an interpretation for constants κ to *NA* in the above definitions. This is by design, so that the user of the library can choose themselves how many constants their generic representation has. Any types that are not part of the constants, automatically become part of the family of recursive datatypes when the `generics-mrsop` library generates codes using TemplateHaskell. This allows the user to 'limit' the depth of the recursive description of types. For example, a user might want to treat *String*s as atoms, instead of lists of characters. If we make sure *String*s are constants in our generic universe, the *String* type will not be deconstructed further. We define the interpretation of constants by means of a *singleton* type, which relates each constant value to exactly one type-level code for constants [8].

```
data SKonstant :: Konstant → *where
  SName :: Name → SKonstant KName
  SArgs :: Args → SKonstant KArgs
  SInt  :: Int  → SKonstant KInt
```

Even though *Name* is defined as `[Char]` and *Args* as `[[Char]]`, both will be treated as constants in our generic universe and will not be deconstructed further into SoP structures.

As we have already seen, if we mark recursive positions explicitly in *regular* datatypes, we can transform a pattern functor *f* into a recursive datatype by taking the Fixpoint *Fix f* [26]. Swierstra, Azero Alcocer, and Saraiva [30] show that a variant of *Fix* exists for two *bifunctors f* and *g* and this allows us to define a recursive generic representation for families of *two* mutually recursive datatypes. If we extend this pattern to trifunctors f, g, h, which allows for a familiy of *three* mutually recursive datatypes, we see a clear pattern in the types of *Fix* emerge.

$$
\begin{aligned}
&Fix   :: (* \to *) \to *\\
&Fix_2 :: (* \to * \to *) \to (* \to * \to *) \to *\\
&Fix_3 :: (* \to * \to * \to *) \to (* \to * \to * \to *) \to (* \to * \to * \to *) \to *
\end{aligned}
$$

Which, using exponentiation, we can write as

$$Fix_3 :: ((*^3 \to *))^3 \to *$$

Yakushev et al. [33] recognise this pattern, and define a *Fix* that is suitable for a family of *any* number of mutually recursive datatypes. Using $Fix\ n$ as the type with exactly n inhabitants, we can then define Fix_3 as:

$$Fix_3 :: Fin\ n \to (Fin\ n \to ((Fin\ n \to *) \to *)) \to *$$

Reshuffling those arguments, we see that we have generalised a fixed point for endofunctors over $*$ to a fixed point for endofunctors over the indexed set $Fix\ n \to *$ [33].

$$Fix_3 :: ((Fin\ n \to *) \to (Fin\ n \to *)) \to (Fin\ n \to *)$$

However, as we already discussed, we opt to use *Nat* as a coarse representation for *Fin n*, such that we do not have to carry around *SNat n* everywhere. We can then write the following type for *Fix* as:

$$\begin{aligned}&\textbf{data}\ Fix :: ((Nat \to *) \to (Nat \to *)) \to (Nat \to *)\ \textbf{where}\\ &\quad Fix :: f\ (Fix\ f)\ n \to Fix\ f\ n\end{aligned}$$

Though not immediately obvious, Rep_{mrsop}, after reshuffling some arguments, is indeed a mapping from families of types to families of types, signified by the fact that its kind is $(Nat \to *) \to (Nat \to *)$.

$$\begin{aligned}&\textbf{data}\ RepF :: (kon \to *) \to [[[Atom\ kon]]] \to (Nat \to *) \to (Nat \to *)\ \textbf{where}\\ &\quad RepF :: Rep\ \kappa\ \varphi\ (Lkup\ ix\ codes) \to RepF\ \kappa\ codes\ \varphi\ ix\end{aligned}$$

It is also an endofunctor, as we can define a mapping operation that adheres to the functor laws:

$$\begin{aligned}&mapNS\ \ :: (\forall ix.\varphi\ ix \to \chi\ ix) \to NS\ \varphi\ ks\ \ \ \to NS\ \chi\ ks\\ &mapNP\ \ :: (\forall ix.\varphi\ ix \to \chi\ ix) \to NP\ \varphi\ xs\ \ \ \to NP\ \chi\ xs\\ &mapRep :: (\forall ix.\varphi\ ix \to \chi\ ix) \to Rep\ \kappa\ \varphi\ c \to Rep\ \kappa\ \chi\ c\\ &mapRep\ f = Rep \circ mapNS\ (mapNP\ (mapNA\ f)) \circ unRep\end{aligned}$$

This means, that we can turn the shallow representation Rep_{sop} into a deep (recursive) representation, by taking applying *RepF* to *Fix*:

$$Lkup\ fam\ ix \equiv Fix\ (RepF\ \kappa\ (Code_{mrsop}\ fam))\ ix$$

However, in the rest of this thesis, we will instead use a version of *Fix* that inlines the definition of *RepF* directly into its body, reducing the amount of pattern matching that we need to do when working with recursive datatypes.

$$\begin{aligned}&\textbf{data}\ Fix\ (kon \to *) \to [[[Atom\ kon]]] \to (Nat \to *)\ \textbf{where}\\ &\quad Fix :: Rep\ \kappa\ (Fix\ \kappa\ codes)\ (Lkup\ n\ codes) \to Fix\ \kappa\ codes\ n\end{aligned}$$

Analogous to the non-indexed *Fix*, we can define a generic fold, that allows us to collapse our datatype using a provided F-algebra. Note though, that the values that our fold produce are now indexed over *ix*. That is, we could choose a different return type for each member of the family. In case that that is not desired, we can set $\varphi \equiv K\ a$, such that we return *a* for every possible index *ix*.

$$\begin{aligned}&cata :: (\forall iy.Rep\ \kappa\ \varphi\ (Lkup\ iy\ codes) \to \varphi\ iy) \to Fix\ \kappa\ \varphi\ ix \to \varphi\ ix\\ &cata\ f\ (Fix\ x) = f\ (mapRep\ (cata\ f)\ x)\end{aligned}$$

Also, we can freely convert from a shallow representation to a deep representation using *mapRep*:

$$\begin{aligned}&dfrom :: (Family\ \kappa\ fam\ codes, a \sim Lkup\ fam\ ix) \Rightarrow a \to Fix\ \kappa\ codes\ ix\\ &dfrom = Fix \circ mapRep\ (dfrom \circ unEl) \circ from\end{aligned}$$

We now have the benefits of the combinator-based `generics-sop` approach, but also the benefits of working with a deep representation like in the `regular` library, whilst also supporting families of

mutually recursive datatypes. The expressiveness of `generics-mrsop`'s universe hits a sweet spot for representing ASTs generically. Let us return to our example of *gsize*. With `generics-mrsop`, we can now define *gsize* for *Expr* and friends, without explicitly marking their recursive positions using an auxiliary typeclass *Size*. To do this, we write an algebra that collects integers. Note that we wrap this integer into the constant functor *K*, such that for each member of the family of types, our algebra returns the same type of value.

The implementation of the algebra is succinct, due to the fact that there is a *Num* instance for *K Int a*. This means we can use integer literal syntax to define values of type *K Int a*. We use *elimRep* to collapse a *Rep* into an integer value by providing it the eliminators for constants, recursive positions and products. Constants are easy, as they will never increase the count and thus always eliminate to 0. Recursive positions are a bit trickier. We need to provide an eliminator $\forall iy.K\ Int\ iy \to K\ Int\ ix$ where *iy* is the index over which the *algebra* is indexed, which will vary for each layer of recursion that *cata* goes through, and *ix* is the index of the final return value of *gsize*. Luckily, we can trivially convert any *K a x* to a *K a y*, because the second type parameter of *K* is a phantom parameter. For this, We can define

```
coerce :: K a x → K a y
coerce (K x) = K x
```

Haskell already comes with a more general definition of this function in the *Data.Coerce* module that can convert between any two representationally equal types [3]. Now what is left is to provide an eliminator for *NP*. For this, we can use the Prelude function *sum* :: *Num a* ⇒ [*a*] → *a*, using the fact that *K Int a* has a *Num* instance. We now truly have a boilerplate-free implementations of *gsize*, that works for any family of datatypes.

```
size :: (Family κ fam codes, IsNat ix, a~Lkup ix fam) ⇒ a → Int
size = getK ∘ cata sizeAlg ∘ dfrom
  where
    sizeAlg :: ∀iy.IsNat iy ⇒ Rep κ (K Int) xs → K Int iy
    sizeAlg = (1+) ∘ elimRep (const 0) coerce sum
```

4.2 Pattern matching

Programmers are used to working with datatypes through constructors and *pattern matching*. Just like in the Agda implementation, we can provide the programmer with a *view* on our representation, such that we can talk about values in terms of a constructor and its fields. In our Agda implementation, we defined constructors *Constr* as a function *Constr* : (*s* : *SOP*) → *Fin* (*length s*). However, due to Haskell's lack of dependent types, such a function is not possible to implement. Instead, we define *Constr* as a *relation* between sums and *Nat*.

```
data Constr :: [k] → Nat → ∗where
  CS :: Constr xs n → Constr (x : xs) (S n)
  CZ :: Constr (x : xs) Z
```

We can then view a SoP *sum* as its constructor *n* and its fields *Lkup n sum*

```
data View (κ :: kon → ∗) (φ :: Nat → ∗) (sum :: [[Atom kon]]) :: ∗where
  Tag :: IsNat n ⇒ Constr sum n → NP (NA κ φ) (Lkup n sum) → View κ φ sum
sop :: Rep κ fam sum → View κ fam sum
```

```
sop (Rep rep) = go rep
  where
    go :: NS (NP (NA κ fam)) sum → View κ fam sum
    go (H poa) = Tag CZ poa
    go (T (go → Tag c poa)) = Tag (CS c) poa
```

Dually, given some constructor, and its fields, we can construct a value:

```
inj :: Constr sum n → PoA κ fam (Lkup n sum) → Rep κ fam sum
inj c = Rep ∘ go c
  where
    go :: Constr sum n → PoA κ fam (Lkup n sum) → NS (NP (NA κ fam)) sum
    go CZ poa = Here poa
    go (CS c) poa = There (go c poa)
```

The `generics-mrsop` library generates pattern synonyms for each constructor using *TemplateHaskell*. For example, for our *Expr* datatype, the following pattern synonyms are generated

```
pattern Var_ = CZ
pattern Int_ = CS CZ
pattern Add_ = CS (CS CZ)
```

In combination with the *ViewPatterns* language extension, we can very naturally program with the representation of a datatype, as if it were a normal Haskell value. Say we have a function *nonZero* that counts all *Val*s that are non-zero.

```
nonZero (Int n) = bool (n > 0) 0 1
nonZero (Var x) = 0
nonZero (Add e1 e2) = nonZero e1 + nonZero e2
```

We can write a similar function using *sop* and *ViewPatterns*:

```
nonZero = cata alg
  where
    alg :: Rep κ (K Int) xs → K Int ix
    alg (sop → Tag Int_ (v :* NP0)) = bool (v > 0) 0 1
    alg (sop → Tag _    xs       ) = elimNP sum xs
```

The implementation is very similar to that of the original *nonZero*. However, it has two benefits from the original implementation. First of all, we didn't have to write out recursion explicitly, but could use *cata* to recurse for us. Second of all, note that the second line of *Alg* works for *any* constructor with recursive positions, not just *Add*. This means if we would refactor *Expr* to also have a constructor *Mul e1 e2*, we would not have to change our implementation of *nonZero*. In the following sections, we will often use *ViewPatterns* and the *sop* function when writing code.

4.2.1 Discussion and remaining issues

Though `generics-mrsop` seems to be great in terms of both expressiveness and ease of use, there is one very serious issue that we ran into when developing it. Compilation times and memory usage of GHC when compiling code using `generics-mrsop` seem to grow exponentially with the size of the codes needed to describe the types. This means that in practice, even comparetively small ASTs

like that of the Go language fail to compile because GHC runs out of memory. We have filed this as an issue upstream [1] and hopefully in the future `generics-mrsop` will support larger datatypes. After tracing GHC itself, we found out that most of the time seems to be spent in the exhaustiveness checker, that tries to make sure that our *from* and *to* functions cover all branches. Disabling the exhaustiveness checker has no effect however, as the act of disabling currently only seems to suppress the warning, instead of fully disabling it.[2].

Originally, we wanted to develop a diffing and merging tool that would be applicable to many popular languages. However, this performance regression forced us to readjust our scope. Instead we picked two languages with rather modestly sized ASTs. The first one being Clojure, given it was also used in previous experiments on generic diffing and merging [9]. And the second one being Lua, given it considerably more complex than the Clojure AST, and also actually uses mutual recursion in its AST. Note that even though we have limited our research to only these two languages, the framework that we will present in the coming sections is indeed fully generic over the AST. Once the memory issue is fixed in GHC, it should be possible to apply our code to bigger amount of programming languages.

4.3 Implementing an efficient diffing algorithm

In this section, we will show how we ported *gdiff* edit scripts from Agda to the `generics-mrsop` library. We will also show how the algorithm is adjusted to generate edit scripts efficiently by using dynamic programming. Lempsink, Leather, and Löh [13] already have an implementation for *gdiff* in Haskell, but two things motivate us for a rewrite. First of all, the *gdiff* library is not directly compatible with any generic programming library, but uses its own encoding of generic programming instead. By porting the algorithm to the `generics-mrsop` universe, we can diff any programming language, for which there exists an AST on Hackage. The second reason is that our initial benchmarks that we performed show that the original implementation of *gdiff* is not well-optimised, making it unsuitable for diffing large trees.

4.3.1 Edit script

In the Agda implementation of edit scripts, constructors and constants were considered as the characters of our edit script. For this, it introduced two functions $cof : Atom \to Set$ and $fields : (\alpha : Atom) \to cof\ \alpha \to Prod$ that allowed us to calculate how many new 'characters' an insertion or deletion of a character would introduce on our stacks. In Haskell, writing a dependent function like *fields* is not possible, so instead, we define a relation between *Atom*s and the type of their fields. We distinguish two cases. A constant value $\kappa\ k$ has no fields, and a constructor c has fields $Lkp\ c\ (Lkup\ n\ codes)$.

$$
\begin{aligned}
&\textbf{data}\ Cof\ (\kappa :: kon \to *)\ (codes :: [[[Atom\ kon]]]) :: Atom\ kon \to [Atom\ kon] \to *\ \textbf{where}\\
&\quad ConstrK :: \kappa\ k \to Cof\ \kappa\ codes\ (K\ k)\ [\,]\\
&\quad ConstrI ::\\
&\qquad (IsNat\ c, IsNat\ n) \Rightarrow Constr\ (Lkup\ n\ codes)\ c\\
&\qquad\qquad \to ListPrf\ (Lkup\ c\ (Lkup\ n\ codes))\\
&\qquad\qquad \to Cof\ \kappa\ codes\ (I\ n)\ (Lkup\ c\ (Lkup\ n\ codes))
\end{aligned}
$$

Note, that we also keep around a *singleton* list for the fields. As we will see, we need to carry around this singleton to later be able to apply the edit script correctly. Using this definition, we can define an *SoP* view of an atom that treats constants and recursive positions uniformly.

[1] `https://gitlab.haskell.org/ghc/ghc/issues/14987`

[2] `https://github.com/VictorCMiraldo/generics-mrsop/issues/6`

```
data ViewNA κ φ codes a where
  TagNA :: Cof κ codes a t → PoA κ (AnnFix κ codes φ) t → ViewNA κ φ codes a
sopNA :: NA κ (Fix κ codes) a → ViewNA κ φ codes a
sopNA (NA_K k                          ) = Match (ConstrK k) NP0
sopNA (NA_I (Fix (sop → Tag c poa))) = TagNA (ConstrI c (listPrfNP poa)) poa
```

Using this definition of *Cof* we can now define a type for edit scripts that closely follows the definition of the Agda version.

```
data ES (κ :: kon → ∗) (codes :: [[[Atom kon]]]) :: [Atom kon] → [Atom kon] → ∗where
  ES0 ::                                                          ES κ codes [ ]      [ ]
  Ins  :: Cof κ codes a fields → ES κ codes i              (fields ++ j) → ES κ codes i       (a : j)
  Del  :: Cof κ codes a fields → ES κ codes (fields ++ i) j              → ES κ codes (a : i) j
  Cpy :: Cof κ codes a fields → ES κ codes (fields ++ i) (fields ++ j) → ES κ codes (a : i) (a : j)
```

Recall that for applying an insertion, we required a *split* lemma, that allowed us to pop off the *fields* of our stack, in order to build our atom *a*.

```
insCof :: Cof a fields → PoA (fields ++ xs) → PoA (a : xs)
```

However, Haskell is not dependently typed, so we cannot pattern match on the type-level list *fields* in our implementation of *split*. Instead, *split* requires us to provide a *singleton* list value, that proves that we can indeed construct a list that contains all *fields*.

```
data ListPrf :: [k] → ∗where
  Nil :: ListPrf [ ]
  Cons :: ListPrf l → ListPrf (x : l)
split :: ListPrf xs → NP p (xs ++ ys) → (NP p xs, NP p ys)
split Nil poa = (NP0, poa)
split (Cons p) (x :∗ rs) =
  let (xs, rest) = split p rs
  in  (x :∗ xs, rest)
```

Luckily we defined *Cof* to carry around a singleton list around inside. This is exactly the singleton we need to supply to *split* in order to implement *insCof*

```
insCof (ConstrI c sfields) xs =
  let (fields, xs′) = split sfields xs
  in NA_I (Fix $ inj c fields) :∗ xs′
insCof (ConstrK k) xs = NA_K k :∗ xs
```

The implementation of *delCof* and *applyES* are exactly the same as their Agda counterparts.

The original Haskell implementation by Lempsink, Leather, and Löh [13], was built when Haskell did not have support for *DataKinds* yet. Type-level list constructors were simply of kind ∗ Instead of of kind [*a*].

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

This meant that in order to make edit scripts, and functions operating on edit scripts well typed, the edit scripts did not only have to carry around a singleton list proof for *fields*, but also a singleton list proof for the rest of the stack.

$$Ins :: (Type\ f\ a) \Rightarrow IsList\ flds \to IsList\ tys \to f\ a\ flds \to ES\ f\ txs\ (flds +\!\!+ tys) \to ES\ f\ txs\ (Cons\ a\ tys)$$
$$Del :: (Type\ f\ a) \Rightarrow IsList\ flds \to IsList\ txs \to f\ a\ flds \to ES\ f\ (flds +\!\!+ txs)\ tys \to ES\ f\ (Cons\ a\ txs)\ tys$$

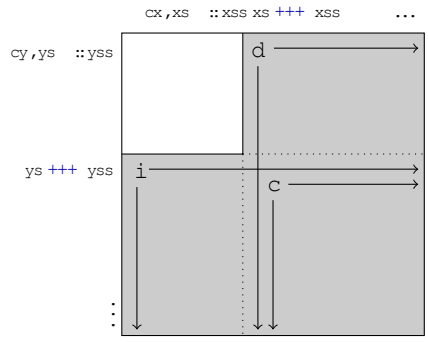
An implementation of *diff* now has to carefully construct and deconstruct these seemingly irrelevant proofs. When we profiled the original *gdiff* implementation, we found out that actually most of the time was spent constructing and deconstructing these seemingly useless singleton lists. Just by using more modern features of GHC, we were able to get a serious gain in performance for free, whilst also simplifying our implementation.

4.3.2 Dynamic programming

As we saw in Chapter 3, the naive approach of generating edit scripts it too slow. To overcome this, Lempsink, Leather, and Löh [13] present a dynamic programming algorithm for generating patches, which we have ported to `generics-mrsop`. The key observation is that the implementation of *diff* performs many sub computations over and over again. By storing these sub computations in a memoization table, subsequent execution steps can reuse previously evaluated results. We can define such a memoization table as follows:

$$\begin{aligned}
&\textbf{data}\ EST\ (\kappa :: kon \to *)\ (codes :: [[[Atom\ kon]]]) :: [Atom\ kon] \to [Atom\ kon] \to *\ \textbf{where}\\
&\quad NN :: ES\ \kappa\ codes\ [\,]\ [\,] \to EST\ \kappa\ codes\ [\,]\ [\,]\\
&\quad NC :: Cof\ \kappa\ codes\ cy\ t \to ES\ \kappa\ codes\ [\,]\ (cy : tys)\\
&\qquad\qquad\qquad\quad \to EST\ \kappa\ codes\ [\,]\ (t +\!\!+ tys)\\
&\qquad\qquad\qquad\quad \to EST\ \kappa\ codes\ [\,]\ (cy : tys)\\
&\quad CN :: Cof\ \kappa\ codes\ cx\ t \to ES\ \kappa\ codes\ (cx : txs)\ [\,]\\
&\qquad\qquad\qquad\quad \to EST\ \kappa\ codes\ (t +\!\!+ txs)\ [\,]\\
&\qquad\qquad\qquad\quad \to EST\ \kappa\ codes\ (cx : txs)\ [\,]\\
&\quad CC :: Cof\ \kappa\ codes\ cx\ t1 \to\\
&\qquad Cof\ \kappa\ codes\ cy\ t2 \to ES\ \kappa\ codes\ (cx : txs)\ (cy : tys)\\
&\qquad\qquad\qquad\quad \to EST\ \kappa\ codes\ (cx : txs)\ (t2 +\!\!+ tys)\\
&\qquad\qquad\qquad\quad \to EST\ \kappa\ codes\ (t1 +\!\!+ txs)\ (cy : tys)\\
&\qquad\qquad\qquad\quad \to EST\ \kappa\ codes\ (t1 +\!\!+ txs)\ (t2 +\!\!+ tys)\\
&\qquad\qquad\qquad\quad \to EST\ \kappa\ codes\ (cx : txs)\ (cy : tys)
\end{aligned}$$

The table describes four different situations, depending on whether the source stack is empty, the target stack is empty, or both are non-empty. The *CC* case is the most interesting. Lempsink, Leather, and Löh [13] provide the following visualization of the *CC* case.



We can either insert *cx*, delete *cy*, or copy if *cx* and *cy* are equal. These three choices lead to three subcomputation *i*, *d* and *c*. As is clear from the image above, both problems *i* and *d* potentially reuse

results from c. This is accomplished by always computing the sub-computation c, whether cx and cy are equal or not.

Building the table is easy for the cases where either the source or the target stack is empty. In case of an empty target stack, the only way we can proceed is by a deletion, and in case of an empty source stack we can only proceed by means of an insertion.

```
diffT o NP0 NP0 = NN ES0
diffT o ((sopNA → TagNA c poa) :∗ xs) NP0 =
  let d = diffT o (appendNP poa xs) NP0
  in  CN c (Del e (getDiff d)) d
diffT o NP0 ((sopNA → TagNA c poa) :∗ ys) =
  let i = diffT o NP0 (appendNP poa ys)
  in  NC c (Ins c (getDiff i)) i
```

The interesting case is when both the source and target lists are non-empty. First we calculate the the diff of the fields of cx and cy. We calculate this result regardless of whether we will emit a *Cpy*. This seems wasteful at first, but as we can see in the image above, there is a big chance that the i and d subcomputations will reuse parts of this computation c. By reusing this subresult, we are able to eliminate two *diffT* calls that were present in the original algorithm. Next, *bestDiffT* will either extend d with a *Del cx* extend, i with a *Ins cy* or extend c with a *Cpy*. The edit script with the lowest cost is then returned.

```
diffT (x@(sopNA → TagNA cx fieldsx) :∗ xs) (y@(sopNA → TagNA cy fieldsy) :∗ ys) =
  let i = extendi cx c
      d = extendd cy c
      c = diffT o (appendNP fieldsx xs) (appendNP fieldsy ys)
      es = bestDiffT cx cy i d c
  in  CC cx cy es i d c
```

The *extendi* and *extend* helper functions take the shared subresult c, and extend it with an extra row or column. Let us look at the case of *extendi*. The type is enlightening. We know that our sub-computation c consumes all the fields t of atom a. Then, we extend this sub-computation c such that it is an edit script that operates on the atom itself.

```
extendi
  :: (Eq1 κ, TestEquality κ)
  ⇒ Cof κ codes a t
  → EST κ codes (t ++ xs) ys
  → EST κ codes (a : xs) ys
extendi cx i@(CN _ d _) = CN cx (Del cx d) i
extendi cx i@(NN d) = CN cx (Del cx d) i
extendi cx d@NC {} =
  case extracti d of
    IES cy c →
      let i = extendi cx c
      in CC cx cy (bestDiffT cx cy i d c) i d c
extendi cx d@CC {} =
  case extracti d of
    IES cy c →
      let i = extendi cx c
      in CC cx cy (bestDiffT cx cy i d c) i d c
```

The cases for *NN* and *CN* are trivial. Because the target stacks are empty, it means we are at the bottom row in the table. Thus we need to use *CN* to extend the edit script with a *Del* operation. In the case that there are rows in the table, we want to add a cell to the left of each row. We use the helper function *extracti* to pop off the row from *d*, and then extend this row with an extra cell *CC*.

```
data IES κ codes a xs ys where
  IES :: Cof κ codes a t → EST κ codes xs (t ++ ys) → IES κ codes a xs ys
extracti :: EST κ codes xs (a : ys) → IES κ codes a xs ys
extracti (CC _ c _ i _ _) = IES c i
extracti (NC c _ i) = IES c i
```

The type of *extendd* is analogous to the type of *extendi*. Its implementation is also analogous. Except it works on the rows of the table instead of the columns of the table.

```
extendd :: Cof κ codes y t → EST κ codes xs (t ++ ys) → EST κ codes xs (y : ys)
```

We omit its implementation as it is very similar to that of *extendi* too.

4.4 Structured Diffs

Porting the structured diff in Section 3.2, from Agda to Haskell is again rather mechanical and straightforward. Instead of showing the entire Haskell implementation, we will show what needs to change in order to support diffs over families of mutually recursive datatypes. The most notable change is that a patch is now a relation between two types inside a family of types, because a patch might change the underlying type of a value. All the subsequent changes to the patch type arise from this change. Furthermore, some changes to the diff datatypes need to be made due to Haskell's lack of dependent types.

4.4.1 Spine

First thing to note, is that just like our port of *ES* to Haskell, we will have to carry around κ and *codes* as type parameters everywhere. In Agda, this can be accomplished elegantly with module parameters, but in Haskell we will have to unfortunately repeat ourselves in all our datatypes. In cases where the values of κ and *codes* are irrelevant, we will omit them for brevity. Second of all, in the Haskell implementation we do not have to pass in *At* and *Al* as arguments to *Spine*, but can instead refer to them directly, because we can define multiple datatypes that are mutually recursive within the same Haskell module.

```
data Spine (κ :: kon → ∗) (codes :: [[[Atom kon]]]) :: [[Atom kon]] → [[Atom kon]] → ∗where
  Scp  :: Spine κ codes s1 s1
  SCns :: Constr s1 c1
       → NP (At κ codes) (Lkup c1 s1)
       → Spine κ codes s1 s1
  SChg :: Constr s1 c1
       → Constr s2 c2
       → Al κ codes (Lkup c1 s1) (Lkup c2 s2)
       → Spine κ codes s1 s2
```

The definitions for *Scp* and *SCns* stay exactly the same. However the definition of *SChg* is slightly different. When dealing with a family of datatypes, a change of constructor means that the type of the

value may actually change. For example, a boolean *expression* could be changed into an `if`-*statement*. The Agda implementation was abstracted over a single sum type, but our Haskell implementation must actually allow for changing the underlying sum-type of the spine. Hence *Spine* is a *relation* on sums. In order to apply the spine, we need to make sure that in the case of *SCns* the source and target types of the patch are the same, otherwise the patch will not apply. For this, we need to pass the source and target index of the patch as *singleton* values [8], such that we can pattern match on them to check them for equality.

```
applySpine :: SNat ix
           → SNat iy
           → Spine κ codes (Lkup ix codes) (Lkup iy codes)
           → Rep κ (Fix κ codes) (Lkup ix codes)
           → Maybe (Rep (Fix κ codes) (Lkup iy codes))
applySpine ix iy (SCns c1 dxs) (sop → Tag c2 xs) = do
  Refl ← testEquality ix iy
  Refl ← testEquality c1 c2
  inj c2 ⟨$⟩ (mapNPM applyAt (zipNP dxs xs))
```

Alignments are the same as the Agda implementation, except for being parameterized over κ and *codes*, and not explicitly taking *At* as a parameter.

4.4.2 Atoms

The type for changes in atoms is no longer parameterized over a recursive value *PatchRec* but instead directly calls *Alμ* itself. Also note that *At* does not change the underlying type of the patch.

```
data At (κ :: kon → ∗) (codes :: [[[Atom kon]]]) :: Atom kon → ∗where
  AtSet :: (κ kon, κ kon) → At κ codes (K kon)
  AtFix :: (IsNat ix) ⇒ Alμ κ codes ix ix → At κ codes (I ix)
```

4.4.3 Tying the recursive knot

Just like changing a constructor can now change the type on which the patch operates, inserting or peeling off a constructor can also change the type. Hence, we will slightly adjust the definition of *Ctx* to account for this fact. If the hole produces a type *iy*, than the *Ctx* itself should produce a type *iy* as well.

```
data Ctx (κ :: kon → ∗) (codes :: [[[Atom kon]]])
  (almu :: Nat → Nat → ∗) (ix :: Nat) :: [Atom kon] → ∗where
  H :: IsNat iy ⇒ almu ix iy              → PoA κ (Fix κ codes) xs → Ctx κ codes p ix (I iy : xs)
  T ::            NA κ (Fix κ codes) a → Ctx κ codes p ix xs       → Ctx κ codes p ix (a : xs)
```

Furthermore, insertion and deletions contexts need to be handled slightly differently. When we have a *Ctx iy* that signifies a deletion, it means we will *peel off* that *iy* to expose some existential *ix*. Dually, when we have a *Ctx iy* that signifies an insertion, it means we will insert a patch that *produces* an *iy* for some existential *ix*. To encode this duality, we define a version of *Alμ* with its arguments flipped

```
newtype Alμ⁻ κ codes ix iy = Alμ⁻ (Alμ κ codes iy ix)
```

With this, we can define the notion of an insertion and a deletion context, and give the type of *Alμ*, which is a relation on family indices.

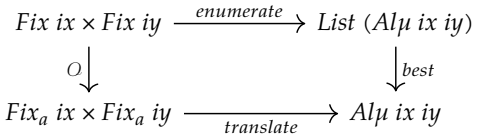


Figure 4.3: enumerating all patches versus choosing a patch through an oracle

```
type InsCtx κ codes = Ctx κ codes (Alμ κ codes)
type DelCtx κ codes = Ctx κ codes (Alμ⁻ κ codes)
data Alμ (κ :: kon → ∗) (codes :: [[[Atom kon]]]) :: Nat → Nat → ∗where
  Spn :: Spine κ codes (Lkup ix codes) (Lkup iy codes)                → Alμ κ codes ix iy
  Ins :: Constr (Lkup iy codes) c → InsCtx κ codes ix (Lkup c (Lkup iy codes)) → Alμ κ codes ix iy
  Del :: Constr (Lkup ix codes) c → DelCtx κ codes iy (Lkup c (Lkup ix codes)) → Alμ κ codes ix iy
```

Applying and deleting insertions and deletion contexts is analogous to the Agda. Applying an insertion context surrounds the recursive value *x1* in a list of fields, and then applies the rest of the patch on *x1*, and applying a deletion context picks a recursive position *x* out of a list of fields, and then transforms it.

```
insCtx :: (IsNat ix, Eq1 κ) ⇒ InsCtx κ codes ix xs → Fix κ codes ix → Maybe (PoA κ (Fix κ codes) xs)
insCtx (H x x2) x1 = (λx → NA_I x :∗ x2) ⟨$⟩ applyAlμ x x1
insCtx (T x x2) x1 = (x :∗) ⟨$⟩ insCtx x2 x1
delCtx :: (Eq1 κ, IsNat ix) ⇒ DelCtx κ codes ix xs → PoA κ (Fix κ codes) xs → Maybe (Fix κ codes ix)
delCtx (H spu atmus) (NA_I x :∗ p) = applyAlμ (unAlmuMin spu) x
delCtx (T atmu al)   (at :∗ p)     = delCtx al p
```

With those two, we can define *applyAlμ*. Note that we pass singleton *SNat ix* and *SNat iy* values to *applySpine*.

```
applyAlμ
  :: ∀κ codes ix iy.(IsNat ix, IsNat iy, Eq1 κ)
  ⇒ Alμ κ codes ix iy → Fix κ codes ix → Maybe (Fix κ codes iy)
applyAlμ (Spn spine) (Fix rep)   =
  Fix ⟨$⟩ applySpine (getSNat @ix Proxy) (getSNat @iy Proxy) spine rep
applyAlμ (Ins c ctx) f@(Fix rep) = Fix ∘ inj c ⟨$⟩ insCtx ctx f
applyAlμ (Del c ctx) (Fix rep)   = delCtx ctx ⋘ match c $ rep
```

4.5 An efficient algorithm for structured diffs

Motivation

The *gdiff* algorithm provides us a quadratic algorithm for calculating edit scripts between datatypes. However, when merging edit scripts, it is hard to figure out what parts of each edit script should be reconciled. This is because edit scripts do not encode structure of the trees on which they operate.

We have shown that the structured patches by Miraldo, Dagand, and Swierstra [21] can be extended to support families of mutually recursive datatypes, giving us a patch structure that does have an easy merge functions. However, no efficient algorithm exists yet to generate these patches. Instead, the define a algorithm that non-deterministically *enumerates* all possible diffs between a

source and destination tree. From this list of patches, the *best* tree is then picked. This is the upper path in the commutative diagram in Figure 4.3. Not only is enumeration very slow, answering the question what constitutes as a best patch is not easy either. One could argue that the patch with the biggest *domain* is the best, but this is very much an *extensional* statement.

Miraldo [20] conjectures that *gdiff* can be used as an *oracle* that drives an algorithm for generating structured patches for regular datatypes. We use *gdiff* to annotate the source and target tree with extra information, which allows us to then distill a structured patch. See the lower part of Figure 4.3. In this section we show that this idea can be implemented in Haskell to get a quadratic algorithm for calculating structured patches whilst also supporting mutually recursive datatypes.

4.5.1 Structured patches from annotated trees

The main idea behind the approach of Miraldo [20] is the following. If we know from the source tree what parts are kept and modified, and from the target tree what parts are kept and modified, then by traversing them in parallel we can distill a structured patch from this information. It is then the task of some sufficiently efficient oracle (in our case, the *gdiff* algorithm) to provide these annotations on the source and destination trees. To illustrate this approach, let us introduce the simple datatype of two-three-trees and two inhabitants of that tree.

```
data TTT = Two Int TTT TTT
         | Three Int TTT TTT TTT
         | Leaf
t1, t2 : Fix CodesTTT Z
t1 = dfrom $ Three 1 Leaf (Two 2 Leaf Leaf) (Two 3 Leaf Leaf)
t2 = dfrom $ Three 1 Leaf (Two 2 Leaf Leaf) (Three 3 Leaf Leaf Leaf)
```

Let us now define a variation of *Fix* that carries an annotation at each recursive position and a type for annotations that marks positions as either copied or modified. The reader might recognise this as the cofree comonad over functors $f :: (Nat \to *) \to (Nat \to *)$.

```
data AnnFix codes (φ :: Nat → ∗) (ix :: Nat) =
  AnnFix (φ ix) (Rep (AnnFix codes φ) (Lkup ix codes))
data Ann = Copy | Modify
```

The oracle then provides us with two functions. One that will annotate our source tree t_1 and one that will annotate our destination tree t_2. Then given two of these annotated trees, we can calculate these to a structured patch between t_1 and t_2 using the function *translate*. In figure 4.4, we see the annotated versions of our trees t_1', t_2', and the result of calling *translate* on those trees. Important to note here is that we want the yet to be defined *translate* function to operate in $O(n)$ time. We already hinted that our oracle will be implemented in terms of the *gdiff* algorithm, which has a time complexity of $O(n^2)$. Because $O(n^2) + O(n) = O(n^2)$, we can freely then combine *translate* and *gdiff* to end up with an algorithm that is still quadratic, but generates structured patches instead of edit scripts.

```
translate :: AnnFix codes (K Ann) ix → AnnFix (K Ann) iy → Alμ ix iy
t1', t2' :: AnnFix CodesTTT (K Ann) Z
t1' = oracleAnnotateSource t1
t2' = oracleAnnotateSource t2
d14 :: Alμ CodesTTT Z Z
d14 = translate t1' t2'
```

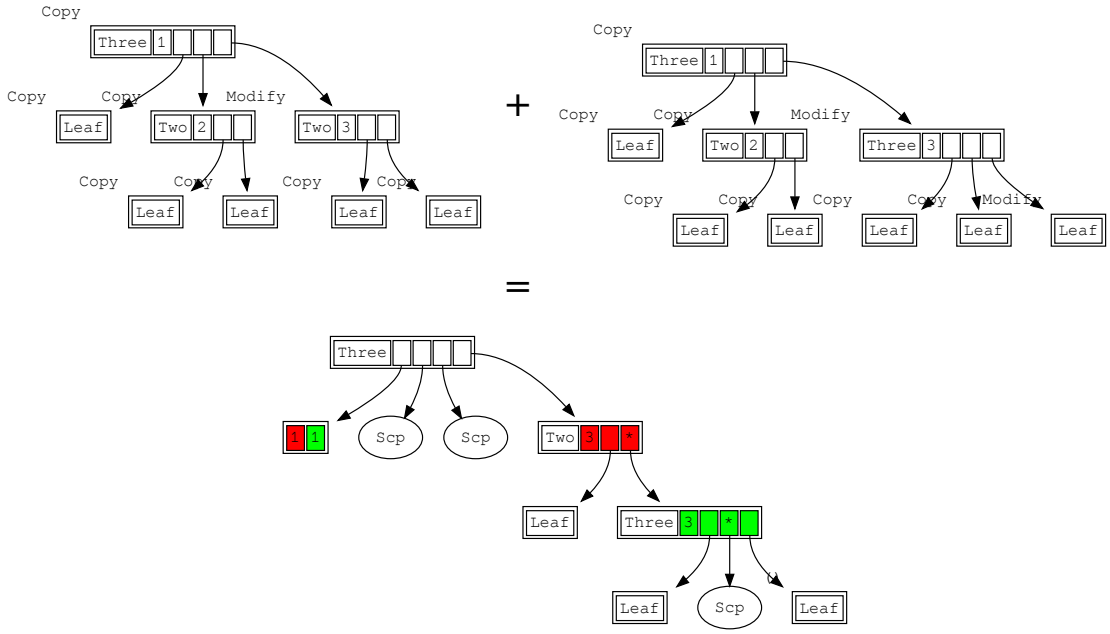


Figure 4.4: Given we know from both the source and the target tree which parts are kept and which parts are modified, we can create a structured patch in $O(n)$

4.5.2 Implementing *translate*

The *translate* function starts by traversing the two trees pair-wise from the top. If both the left-hand side and the right-hand side are annotated by a *Copy*, we know that we will have to emit a *spine*.

```
translate :: AnnFix Ann ix → AnnFix Ann iy → Alμ ix iy
translate x@(AnnFix Copy rep1) y@(AnnFix Copy rep2) =
  Spn (translateSpine (getSNat @ix) (getSNat @iy) rep1 rep2)
```

Spine

In Figure 4.4, we see that the toplevel recursive position has a *Copy* annotation in both trees, and thus indeed a spine is emitted. In this case, an *SCns* is then emitted by *translateSpine*, as the toplevel constructor in both t_1 and t_2 is equal. We then zip over the fields of t_1 and t_2, and for each pair of fields, we call *translate* recursively. Given that the first few fields of t_1 and t_2 are all annotated with *Copy*, these recursive calls will emit an *Scp* node, indicating that at those positions the fields are simply copied from the source tree to the target tree. A good patch will try to have as many *Scp* nodes as possible, as they increase the domain of the patch. An *Scp* matches any node and copies it over verbatim. The more copies a patch contains, the higher the chance we will sucessfully merge it.

The code for *translateSpine* is defined below. In order for *translateSpine* to work for mutually recursive types, we need access to the type indices *ix* and *iy*. When the types do *not* match, the only way to emit a spine is by means of a change in constructor (such that the old constructor lives in *ix*, and the new constructor lives in *iy*). Note however, that in our toy example, *ix* and *iy* will always be equal, as *TTT* is not mutually recursive. If the two trees *are* part of the same type, then we check whether the constructors are equal. If they are equal, we proceed by emitting an *SCns*, as already

described above. If they are not equal, we proceed the same way as we would when *ix* and *iy* are not equal. We emit an *SChg*, and have to recursively produce an alignment *Al* between the fields of the two values.

```
translateSpine
  ::  ∀codes ix iy.
      SNat ix
  →   SNat iy
  →   Rep (AnnFix Ann) (Lkup ix codes)
  →   Rep (AnnFix Ann) (Lkup iy codes)
  →   Spine (Lkup ix codes) (Lkup iy codes)
translateSpine six siy s1@(sop → Tag c1 p1) s2@(sop → Tag c2 p2) =
  case testEquality six siy of
    Just Refl →
      if (eq1 `on` mapRep forgetAnn) s1 s2
        then Scp
        else case testEquality c1 c2 of
          Just Refl →
            SCns c1 (mapNP (λ(a × b) → translateAt a b) (zipNP p1 p2))
          Nothing → SChg c1 c2 (translateAl p1 p2)
    Nothing → SChg c1 c2 (translateAl p1 p2)
```

Alignments

In the original implementation of diffing by Miraldo, Dagand, and Swierstra [21], alignments were a difficult thing to compute. Their strategy was to non-deterministically enumerate all options, and then choose the best one. However, now that we have the information about what parts of the source and what parts of the target trees are copied an kept, generating alignments becomes trivial.

```
translateAl :: PoA (AnnFix Ann) xs → PoA (AnnFix Ann) ys → Al xs ys
```

To produce an alignment we traverse the two lists in parallel. If both heads of the list are a *Copy*, an *AX x* is emitted and we recurse on the rest of the fields. If the left-hand side is a *Copy* and the right-hand side is a *Modify*, we emit an insertion *AIns x*. Dually, if the left-hand side is a *Modify*, we insert a *ADel x*. If either the left-hand side or the right-hand side is empty, we pad the alignment with *AIns* or *ADel* respectively to compensate. An example of a produced alignment can be seen in Figure 4.5.

Contexts

In the case that source tree is annotated with a *Copy* and the destination tree is annotated with a *Modify*, we will produce an insertion context. However, to which part of the tree do we make the context point at? We want to point to a field that maximises the chance that traversing it with out source patch *x* will emit an *Scp*, because copies increase the domain of our structured patch, making it more likely to merge. Note however, that this is a greedy heuristic. Making a choice now might affect the amount of reuse further down the tree, so this method is not guaranteed to give us the best patch possible. If we have such a function, named *pointToMaxCopies*, we can define *translate* as follows:

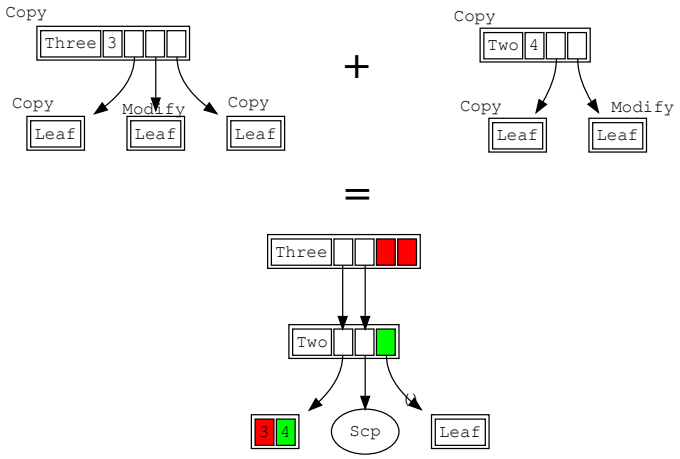


Figure 4.5: Generating an alignment

```
translate :: AnnFix Ann ix → AnnFix Ann iy → Alμ ix iy
translate x@(AnnFix Copy _)                    y@(AnnFix Modify (sop → Tag cy ys))
  | countCopies y > 0 = Ins cy (pointToMaxCopies CtxIns x ys)
  | otherwise         = stiff x y
translate x@(AnnFix Modify (sop → Tag cx xs)) y
  | countCopies x > 0 = Del cx (pointToMaxCopies CtxDel y xs)
  | otherwise         = stiff x y
```

The function

```
countCopies :: AnnFix Ann ix → Int
```

recursively counts the number of *Copy*s that a given tree contains. The *pointToMaxCopies* function will point out exactly *one* recursive position with the most copies. However *pointToMaxCopies* is partial. If there are no recursive positions inside *ys* then there is nothing to point to. But because we assert that *countCopies* $y > 0$, we are sure that when we call *pointToMaxCopies* there is at least *one* recursive position with copies in *ys*. If there are no copies in *y*, we generate a *stiff* patch from *x* to *y*. A stiff patch is a patch that totally fixes the domain of *x* and then produces *y*. It's a patch that has no reuse or merge-potential whatsoever.

Let us define two helper functions in order to define *pointToMaxCopies*. First, we have

```
maximumOn :: Ord k ⇒ (∀x.f x → k) → NP f xs → NS f xs
```

which gives us the maximum element in a list. The type*NS* can be seen as an injection into *NP*. It not only tells us the maximum value, but also tells us where it is located inside the list. Given such a location and a list of fields we can then punch a hole in that list of fields at that location to produce the context. Note that we can use *punchHole* to both produce insertion and deletion contexts. The only difference between the two is whether we continue with *translate x y* or *translate y x*.

```
punchHole :: NS f ys → PoA Fix ys → Fix Ann ix → InsOrDel almu → Ctx almu ix ys
punchHole (Here _)   (NA_K _ :* _)  _ _       = error "we never reach this case."
punchHole (Here _)   (NA_I y  :* ys) x CtxIns  = H (translate x y) ys
punchHole (Here _)   (NA_I y  :* ys) x CtxDel = H (Alμ⁻ (translate y x)) ys
punchHole (There xs) (y       :* ys) x iod     = T y (punchHole xs ys x iod)
```

We can then combine *maximumOn* and *punchHole* to define *pointToMaxCopies*, which punches a hole at the location with the most copy potential.

```
pointToMaxCopies :: PoA (AnnFix Ann) xs → AnnFix Ann ix → InsOrDel almu → Ctx almu ix xs
pointToMaxCopies = punchHole (maximumOn countCopies)
```

4.5.3 Making *translate* efficient

Calculating the amount of copies each subtree has over and over again makes the *translate* function prohibitively slow. To solve this, we perform a prepreocessing step over our annotated trees, that augments each recursive position with an extra annotation that tells how many copies there are in the children of that node.

For this can define a function *synthesize*, which given an algebra annotates each recursive position of the tree with the value of the algebra at that point. If *cata* is a generalisation of *foldr*, then *synthesize* is a generalisation of *scanr*. It is also similar to the notion of *synthesized attributes* in attribute grammars [11].

```
synthesize :: ∀κ φ codes ix
            . (∀iy.χ ix → Rep κ φ (Lkup iy codes) → φ iy)
           → AnnFix κ codes χ ix
           → AnnFix κ codes φ ix
synthesize alg =
  cata $ λann xs → AnnFix (f ann (mapRep (λ(AnnFix a _) → a) xs)) xs
```

We then define an algebra that counts the number of copies in a node.

```
copiesAlgebra :: K Ann ix → Rep (K Int) xs → K Int ix
copiesAlgebra (K Copy)   = (1+) ∘ elimRep 0 coerce sum
copiesAlgebra (K Modify) =       elimRep 0 coerce sum
```

And then use *synthesize* to get a version of our tree that tells whether it is a *Copy* or a *Modify* but also tells us how many of its children have *Copy* nodes.

```
withCopies :: AnnFix Ann ix → AnnFix (Int × Ann) ix
withCopies = synthesize (copiesAlgebra × const)
```

Now we can replace our *countCopies* function which was $O(n)$ with a version that is $O(1)$:

```
countCopies (AnnFix (c × _) _) = c
```

With that, *translate* is now $O(n)$ as well.

4.5.4 Using gdiff to annotate trees

To use the *gdiff* algorithm as an oracle to generate annotations, we first produce an edit script between the source and target tree. We then traverse the edit script two times. One time, we traverse it together with the source tree, and the second time we traverse it with the target tree, leaving us with two annotated trees. Let us look at the case of traversing the source tree first. The base case is trivial.

```
annSrc' :: PoA Fix xs → ES xs ys → PoA (AnnFix (K Ann)) xs
annSrc' NP0 ES0 = NP0
```

Then, when we encounter a deletion, we annotate the node x as *Modify*. As a deletion is performed by modifying the source tree to get a value for the target tree. And if we encounter a copy edit operation, we annotate x with a *Copy*. Note that our usage of *fromJust* is safe here as we assume that the edit script that we feed to *annSrc* is the edit script that was generated from diffing *xs* and *ys*. Here *insCofAnn* is a variation of *insCof* that allows us to set the annotation in the case the atom is an *NA_I*.

```
annSrc' (x :* xs) (Del _ c es) =
  let poa = fromJust $ matchCof c x
  in  insCofAnn c (K Modify) (annSrc' (appendNP poa xs) es)
annSrc' (x :* xs) (Cpy _ c es) =
  let poa = fromJust $ matchCof c x
  in  insCofAnn c (K (annSrc' (appendNP poa xs) es)
```

Insertions do not change the source tree in any way, and hence when we encounter one, we simply skip it and continue traversing the tree and the edit script.

```
annSrc' (x :* xs) (Ins _ c es) = annSrc' (x :* xs) es
```

Dually we define

```
annDest' :: PoA Fix ys → ES xs ys → PoA (AnnFix (K Ann)) ys
```

which has exactly the same implementation as *annSrc'* but skips deletions, and emits *Modify* when ecountering insertions. This is because a deletion will not require anything from the target tree at all, but an insertion does require something from the target tree. Note that the type of *annSrc'* operates on the *source stack xs*, whilst *annDest'* operates on the *target stack ys*. No special care has to be taken to support mutual recursion, as the edit scripts already operate on families of mutually recursive datatypes.

We can then calculate our annotated trees t'_1 and *t1y* using *annSrc'* and *annDest'*, yielding the trees in Figure 4.4, and then feed the annotated trees into *translate* to get our structured patch.

```
es :: ES [I Z] [I Z]
es = gdiff t1 t2
t'1, t'2 :: AnnFix CodesTTT (K Ann) Z
t'1 = annSrc'  (t1 :* NP0) es
t'2 = annDest' (t2 :* NP0) es
d12 :: Alμ Z Z
d12 = translate (countCopies t'1) (countCopies t'2)
```

Discussion

Note that the entire algorithm still operates in $O(n^2)$ time. Generating the edit script takes $O(n^2)$. Then using the edit script to annotated the source and target tree takes another $O(n)$. Then counting the amount of copies on the source and target tree also takes $O(n)$. Then translating these annotated trees to structured patches also takes $O(n)$. In total, this gives us a complexity of $O(n^2) + 3O(n) = O(n^2)$. We have thus successfully implemented a quadratic algorithm for generating structured patches. However, there is a caveat. We made the choice to always pursue the path that maximizes the amount of copies in that path. However, we do not know whether this is the best strategy to generate the best structured patches. We already mentioned that reasoning about what constitutes

a "good" patch is hard. In Chapter 5 we will perform an empirical evaluation on real-world data, to see whether the patches we generate are good enough to be mergeable. Miraldo [20] provides a proof for regular types that this method of generating structured patches produces well-behaved patches. Formally, that is

$$apply\ (diff\ source\ dest)\ source \equiv dest$$

We did not prove whether our implementation, which operates on mutually recursive types instead, also has this property. However, we did check whether this property holds using QuickCheck. Also, we checked this property on all mined data that was used in Chapter 5. This gives us confidence that our implementation is indeed correct.

4.6 Merges

In the Agda implementation of merges that was presented in Chapter 3, the *merge p q* function that incorporates the changes *p* into the changes of *q* was *total* if we had a proof that *p* and *q* were disjoint. However, due to Haskell's lack of dependent types, we can not carry around the disjointness proof to make *merge* a total function. Instead, we adjust the type of *merge* to return a *Maybe Alμ*.

```
mergeAlμ :: Alμ ix iy → Alμ ix iy → Maybe (Alμ ix iy)
```

When two patches are not disjoint, we return *Nothing*, otherwise, we apply the change *p* onto the changes of *q*, similarly to how we would apply a patch to a tree. Furthermore, we have to account for the fact that we are now merging mutually recursive types instead of regular types. This means we have to expand on what we consider to be disjoint patches. We will highlight those places where this matters in the following subsections.

4.6.1 Merging fixpoints

Merging at the recursive level is the most intricate. First of all, let us handle the easy cases. Namely, a patch is never disjoint from itself, and hence we can never reconcile an insertion with an insertion, or a deletion with a deletion.

```
mergeAlμ Ins {} Ins {}  = Nothing
mergeAlμ Del {} Del {} = Nothing
```

As we already know, insertions are always allowed, as they do not restrict the domain of the patch in any way. We want to apply the insertion to a patch, instead of a tree. The insertion would have caused an extra constructor to appear on the destination tree, so to our new patch we add an extra constructor *c1*. However, an *SCns* is of type *Alμ ix ix*. It does not change the type of the patch. Hence, we can only do this if we assert that *ix*∼*iy*. Or in other words, we can only do this merge if insertion did not cause the destination type of the patch to change.

```
mergeAlμ (Ins c₁ s₁) y = do
  Refl ← testEquality (getSNat @ix Proxy) (getSNat @iy Proxy)
  Spn ∘ SCns c₁ ⟨$⟩ mergeCtxAlμ s₁ y
```

Dually, any patch can be applied to an insertion, by simply wrapping the resulting patch with the said insertion.

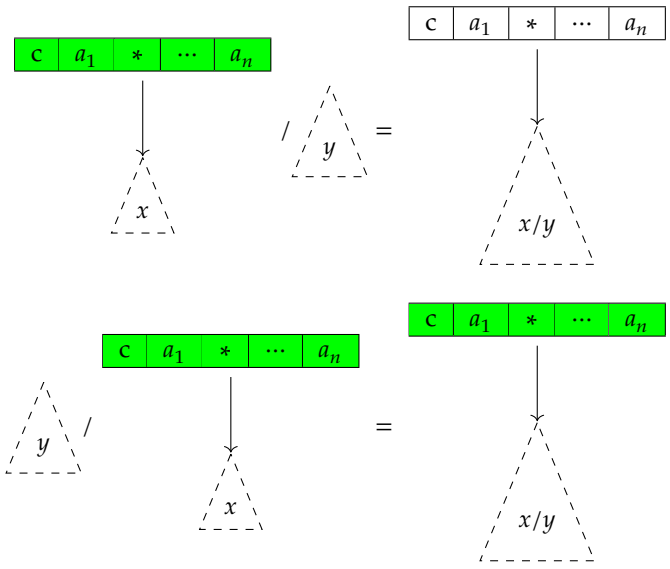


Figure 4.6: Merging insertions

$$mergeAlμ\ x\ (Ins\ c_2\ s_2) = Ins\ c_2\ \langle\$\rangle\ mergeAlμCtx\ x\ s_2$$

The type of *mergeAlμCtx* is as follows.

$$mergeAlμCtx :: Alμ\ \kappa\ codes\ ix\ iy \to InsCtx\ \kappa\ codes\ ix\ xs \to Maybe\ (InsCtx\ \kappa\ codes\ ix\ xs)$$

The *mergeAlμCtx* function takes the patch $Alμ\ ix\ iz$ to which the insertion context points, and then tries to merge it with the provided patch $Alμ\ ix\ iy$. Important to note here, is that the insertion context points to some patch $Alμ\ ix\ iz$ where z is *existential*. We do not know what type this iz points to yet. Hence, we have to explicitly check that $iy{\sim}iz$. Because we can only merge the patch with the hole if they both have the same source and destination type. Again, this is an additional constraint that the original implementation of disjointness for regular types did not have. In Figure 4.6, we see a visual representation of these two cases.

There are two cases where a deletion is disjoint with another patch. First of all, a deletion is disjoint from a copy. Intuitively, this makes sense. If you delete a part from a patch that matches anything, what is left is still a patch that matches anything. In Figure 4.7 we see a graphical depiction of this scenario.

Second of a all, a deletion is disjoint from an *SCns* if they happen to operate on the same constructor. To merge the two, we will have to merge the fields of the deletion context with the fields of the constructor, and make sure that we can merge the patch to which the context points with the corresponding patch in the fields of the *SCns*.

$$\begin{aligned}
&mergeAlμ\ (Del\ c_1\ delCtx)\ (Spn\ (SCns\ c_2\ fields)) = \mathbf{do}\\
&\quad Refl \leftarrow testEquality\ c_1\ c_2\\
&\quad mergeCtxAts\ delCtx\ fields\\
&mergeAlμ\ (Spn\ (SCns\ c_1\ at_1))\ (Del\ c_2\ s_2) = \mathbf{do}\\
&\quad Refl \leftarrow testEquality\ c_1\ c_2\\
&\quad Del\ c_1\ \langle\$\rangle\ mergeAtsCtx\ at_1\ s_2
\end{aligned}$$

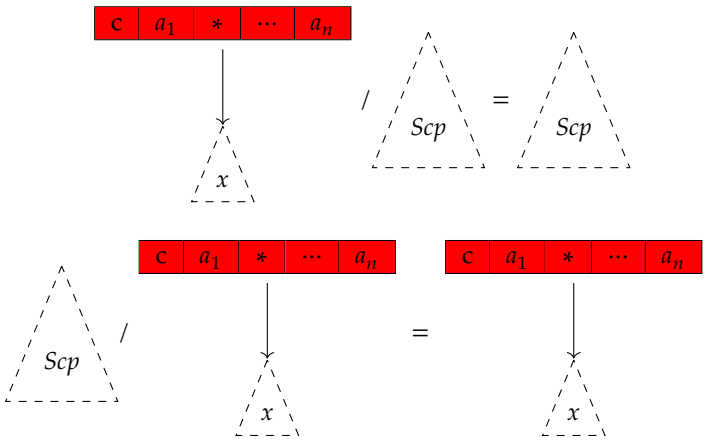


Figure 4.7: *Scp* can be merged with anything, including deletions

The *mergeCtxAts* function overlays the deletion context over the list of fields, and at the place where the hole of the context overlaps with a field, checks that field for disjointness with the hole, and returns the resulting patch of that.

```
mergeCtxAts :: DelCtx iy xs → NP At xs → Maybe (Alμ κ codes ix iy)
```

The implementation is as follows. We distinguish two cases. In the case that we have not yet encountered the hole, we keep traversing.

```
mergeCtxAts (T almu' ctx) (x :∗ xs) = mergeCtxAts ctx xs
```

If we do find the hole, however, we recursively diff the patch at the hole on the left-hand side with the corresponding patch on the right-hand side. Note that this only works if both patches agree on their destination type. That is, the both return a type *iy*. Also, because the source type of the resulting patch is existentially quantified, we must make sure the source type is indeed *ix*. This is again an extra property that our original notion of disjointness on regular types did not have.

```
mergeCtxAts (H (Alμ⁻ almu') rest) (AtFix almu :∗ xs) = do
  Refl ← testEquality (almuDest almu) (almuDest almu')
  x    ← mergeAlμ almu' almu
  Refl ← testEquality (almuSrc x) (getSNat @ix Proxy)
  guard (and $ elimNP identityAt xs)
  pure x
```

Dually, *mergeAtsCtx* projects out the *DelCtx* onto the list of fields, merges the *Alμ* where they overlap, and use that *Alμ* to construct a new *DelCtx*. A graphical depiction of these two dual scenarios can be found in Figure 4.8.

```
mergeAtsCtx :: NP At xs → DelCtx iy xs → Maybe (DelCtx iy xs)
mergeAtsCtx (AtFix almu :∗ xs) (H (Alμ⁻ almu') rest) = do
  Refl   ← testEquality (almuDest almu) (almuDest almu')
  almu'' ← mergeAlμ almu almu'
```

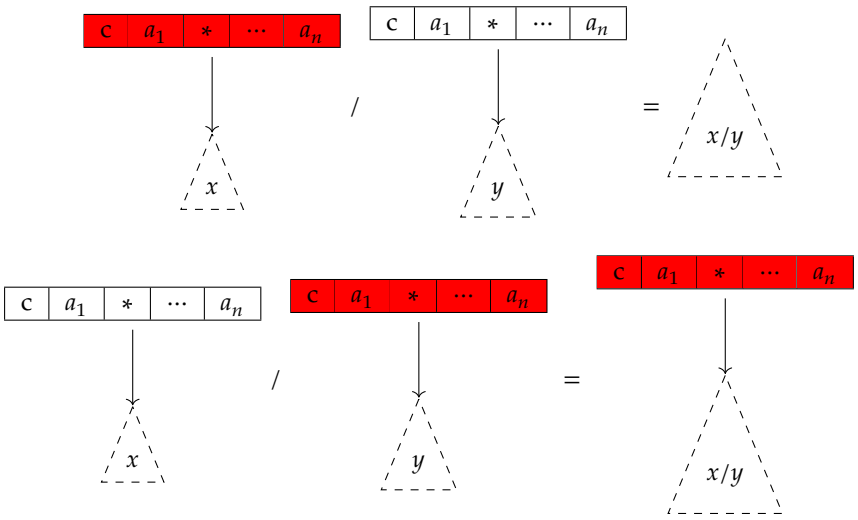


Figure 4.8: Merging deletions with an *SCns*

```
  guard (and $ elimNP identityAt xs)
  pure $ H (Alμ⁻ almu'') rest
mergeAtsCtx (x :* xs) (T a ctx) = do
  T a (<$>) mergeAtsCtx xs ctx
```

In any other case, a deletion is not disjoint and hence can not be merged.

```
mergeAlμ Del {} (Spn SChg {}) = Nothing
mergeAlμ (Spn SChg {}) Del {} = Nothing
```

4.6.2 Merging spines

Spines might be disjoint.

Now, if we have two spines, we follow the coproduct structure, check if they match, and then compare the product values for disjointness.

```
mergeAlμ (Spn s1) (Spn s2) = Spn (<$>) mergeSpine (getSNat @ix) (getSNat @iy) s1 s2
```

The *mergeSpine* function takes the source and target families *ix* and *iy* its first arguments, because we need to inspect them whilst reconciling changes in constructors.

```
mergeSpine :: SNat ix
           → SNat iy
           → Spine κ codes (Lkup ix codes) (Lkup iy codes)
           → Spine κ codes (Lkup ix codes) (Lkup iy codes)
           → Maybe (Spine κ codes (Lkup ix codes) (Lkup iy codes))
```

When either side is a *Scp*, the two spines are trivially disjoint, as an *Scp* matches anything.

```
mergeSpine _ _ Scp s = pure s
mergeSpine _ _ s Scp = pure Scp
```

If the two spines have matching constructors, it means they share a common structure, and we can continue merging. If both sides have an *SCns*, then we thus need to make sure they both operate on the same constructor. And if that is the case, we can simply we can merge their fields pairwise.

```
mergeSpine _ _ (SCns cx xs) (SCns cy ys) = do
  Refl ← testEquality cx cy
  SCns cx ⟨$⟩ mergeAts xs ys
```

The *mergeAts* function traverses the lists in parallel, and calls *mergeAt* for each pair of atoms.

```
mergeAts :: Eq1 κ ⇒ NP (At κ codes) xs → NP (At κ codes) xs → Maybe (NP (At κ codes) xs)
```

A more interesting case, is when the left-hand side does not change the type of the patch (*SCns* :: *Spine ix ix*) and the right-hand side *does* (*SChg*::*Spine ix iy*). We can turn a *Spine ix ix* into a *Spine ix iy*, if both patches agree on the same source constructor, and we can merge the changes of the fields on the left-hand side with the alignment on the right-hand side through *mergeAtAl*.

```
mergeSpine _ _ (SCns cx xs) (SChg cy cz al) = do
  Refl ← testEquality cx cy
  SChg cy cz ⟨$⟩ mergeAtAl xs al
```

Dually, if we have an *SChg* and we want to adapt it with an *SCns* we can do so if the source constructors match. However, this is only possible if the target type of the *SChg* matches the type of the *SCns*. When merging regular datatypes, this is always the case, but because we are merging families of mutually recursive datatypes, we need to perform an extra check to make sure that the *SChg* produces the correct target type. For spines, this is the only difference between the Agda implementation.

```
mergeSpine ix iy (SChg cx cy al) (SCns cz zs) = do
  Refl ← testEquality ix iy
  Refl ← testEquality cx cz
  SCns cy ⟨$⟩ mergeAlAt al zs
```

When both patches change the constructor, they are never disjoint.

```
mergeSpine ix iy SChg { } SChg { } = Nothing
```

4.6.3 Merging alignments

When one of two patches is a change in constructor, whilst the other one is not, we need to adapt the alignment of the one to the changes of the other. This means that we have to check whether an alignment *Al xs ys* is compatible with an *NP At xs*, producing an *NP At ys*, or dually, check whether a list *NP At xs* is compatible with an alignment *Al xs ys*, producing an *Al xs ys*.

```
mergeAlAt :: Al xs ys  → NP At xs → Maybe (NP At ys)
mergeAtAl :: NP At xs → Al xs ys  → Maybe (Al xs ys)
```

When we want to reconcile an insertion with a change of fields, we insert a change in atom *At* that matches the atom *at* trivially. For this we use a helper *makeIdAt* which produces the identity edit for *at*.

$$mergeAlAt\ (AIns\ at\ al)\ xs = (makeIdAt\ at :*) \langle\$\rangle\ mergeAlAt\ al\ xs$$

To reconcile a deletion with our list of changes, the head of our list of changes must be the identity edit. If it is, we pop it off, shrinking the list by one.

$$mergeAlAt\ (ADel\ at\ al)\ (x :* xs) = guard\ isIdentityAt\ x *\rangle\ mergeAlAt\ al\ xs$$

Finally, if the alignment is an AX, we simply zip the two changes at_1 and at_2 and see if they are disjoint pairwise.

$$mergeAlAt\ (AX\ at_1\ al)\ (at_2 :* xs) = (:*) \langle\$\rangle\ mergeAt\ at_1\ at_2 \langle*\rangle\ mergeAlAt\ al\ xs$$

Again, due to the fact that merge is symmetric, we need to implement the opposite case as well. The implementation of *mergeAtAl* is analogous to that of *mergeAlAt*, but instead, we produce an alignment *Al* instead of consuming one.

$$mergeAtAl :: NP\ At\ xs \to Al\ xs\ ys \to Maybe\ (Al\ xs\ ys)$$
$$mergeAtAl\ NP0\ A0 = pure\ A0$$

An insertion can be trivially merged with the list of *At*s, as it leaves the fields untouched. Just like with insertions at the recursive level.

$$mergeAtAl\ xs\ (AIns\ at\ al) = AIns\ at \langle\$\rangle\ mergeAtAl\ xs\ al$$

A change x is only disjoint with a deletion, if x happens to be the identity edit. Otherwise, we have a conflict.

$$mergeAtAl\ (x :* xs)\ (ADel\ at\ al) =$$
$$\quad guard\ identityAt\ x *\rangle\ ADel\ at \langle\$\rangle\ mergeAtAl\ xs\ al$$

And in the case of an AX we zip the two changes together, and see if they elements are disjoint pairwise.

$$mergeAtAl\ (x :* xs)\ (AX\ at\ al) = AX \langle\$\rangle\ (mergeAt\ x\ at) \langle*\rangle\ mergeAtAl\ xs\ al$$

Merging atoms is straightforward. Two constants are disjoint of each other, if and only if one of the two is the identity edit. And if we have two recursive positions, we simply recursively call $mergeAl\mu$.

$$mergeAt :: At\ \kappa\ codes\ a \to At\ \kappa\ codes\ a \to Maybe\ (At\ \kappa\ codes\ a)$$

Discussion

We successfully implemented a simple merging algorithm for structured patches. We had to extend the specification for merges by Miraldo [20] to support families of mutually recursive datatypes. We did not prove that this transformation still adheres to the merge properties from Figure 3.2. But we did use QuickCheck [5] to see whether we could find counter-examples for this property. This turned out to be an instrumental tool to weed out bugs in our implementation. Furthermore, we tested for each merge conflict mined for the experiments in Chapter 5, whether the merge commuted as expected. This gives us confidence our implementation is correct.

5 Experiments

5.1 Data collection

We conducted experiments over a large set of real-world patches, to measure the effectiveness of our approach. We used the same method of collecting data as Garufi [9]. To demonstrate that our approach is truly generic, we collected real-world data for multiple languages. Clojure was chosen because Garufi [9] performed experiments on Clojure as well, and allows us to compare results. Lua was chosen because it is a statement-based language with a much more complex AST than Clojure, whilst still being small enough to not run into memory issues of the GHC compiler. In the future we would like to perform a more comprehensive study with more programming languages, but that is currently blocked on the issues in GHC.

Using the data-mining scripts provided by Garufi [9], we mined twenty popular Clojure and twenty popular Lua repositories for conflicts. Repositories were ranked by a combination of number of collaborators and number of stars. A high number of collaborators hopefully increases our chances of finding merge conflicts, as multiple people are working on the same project at the same time, and a high number of stars might be an indicator of the quality of the code, and hopefully gives a selection of repositories from different domains [9].

For each repository, the history was scraped for commits with two ancestors. So called *merge points*. Let us denote these two ancestor commits as A and B. We can traversed the commit tree upwards until the moment that the two branches A and B diverged, and saved that version of the repository as O. We then performed a three-way merge using `diff3` for each triple of files inside the file trees O A and B. If `diff3` would report a conflict for such a triple of files, we saved that conflict to our dataset.

Note that this approach is not perfect. Some repositories do not use explicit merging as their workflow, but instead *squash* multiple edits into one commit, or *rebase* the history instead of merging. Though in these workflows people will encounter merge conflicts, the git history is being rewritten and from an outside observer it is not possible to find out whether conflicts were present at the time.

Of the tens of thousands of commits mined, we only recovered 1263 conflicts for Lua and 1515 commits for Clojure. However, this is in line with expectations. Merge commits are rare compared to normal commits, and it is even more rare that someone has to manually intervene in such a merge, as postulated by Mens [19]'s 90/10 rule.

5.2 Diff performance

We performed an evaluation of performance of our diffing algorithm. For each pair OA and OB, we measured the time it took to evaluate the produced `gdiff` patch into normal-form. We then stored this measurement, together with the size of the source and target trees. Note that we did not

Language	Solved diffs	Solved *pairs* of diffs by Garufi [9]
Lua	1062 / 1263	x
Clj	1297 / 1515	452 / 616

Table 5.1: Amount of conflicts mined per language. Note that Garufi only counted solved *pairs* of diffs. That is, the cases where both *OA* and *OB* succeeded.

include the conversion to structured patches in the time measurements. For each diff, a memory limit of 12GiB was set. If the memory usage exceeds that point, the diff would be marked as failed. No timeout was set on the diff operation.

In Table 5.1 we show the amount of solved diffs versus the amount of available pairs of *OA* and *OB*. We see that we were able to solve around 85 percent of all diffs. The other cases ran out of memory. Garufi [9] was able to solve around 72 percent of diffs before running out of memory or running out of a 60 second time limit. This shows that our algorithm indeed seems to perform slightly better. No further performance data like running times were collected by Garufi [9] so any other comparison between performance is hard to make.

The maximum observed running time of our diff algorithm is around 25 seconds. At that point, the memoization table would be bigger than 12 GiB. We thought of adding more RAM to our test rig such that we could have a timeout of 60 seconds, just like Garufi [9], however due to memory usage being quadratic, we gave up on that endeavour. Even though, with our average timeout being way lower than that of Garufi [9], we still solved more diffs, which sounds promising.

The theoretical worst case algorithmic complexity of the dynamic programming algorithm for diffing ordered trees is $O\left(n_1 n_2 \min\left(d_1, l_1\right) \min\left(d_2, l_2\right)\right)$, where n_1 and n_2 are the amount of nodes in each tree, d_1 and d_2 are the depth of the tree, and l_1 and l_2 are the number of leaves in the tree [15]. We expect n_1 and n_2 to be similar in size in the majority of cases, because we speculate that most edits in source control are updates to existing code, instead of large insertions or deletions of code. Looking at the histogram in Figure 5.1 indeed shows that in the majority of cases there is only a small difference.

In Figure 5.2, we plotted the amount of nodes in the tree against the running time of the algorithm. The axes are in log-log form, which is convenient because monomial formulas (of the form $y = ax^k$) show up as straight lines with slope k in such plots. We see from the plot, that there is a clear linear trend in the worst-case performance. The slope of this trend is around $k = 2$, which indicates that our observed worst-case performance is within our expectations, namely around $O(n^2)$. Next, we see that our best-case performance also shows a linear trend, with slope $k = 1$. This is also within our expectations, as in the best case scenario, two trees are basically the same, and omit only copies in $O(n)$ time. There are no samples outside of these two trend lines, which is reassuring. It shows that our algorithm is performing as expected.

5.3 Merges

Next, we measured how many merges we could actually complete. For each triplet (A, O, B) we again calculated the *OA* and *OB* patch, and then calculated the merge patch M from those. Because now we have to calculate two patches in parallel, we had to run experiments on a machine with more RAM available. We removed the memory limit from the test-suite, and instead added a two minute timeout to each run. In Table 5.2 we see the results of these experiments. For Clojure, we were able to solve around 15 percent of conflicts that `diff3` could not solve automatically. Whilst for Lua we were able to solve around 13 percent of conflicts automatically.

The last column in the tables, labeled *OOM* indicates the amount of merges that did not complete

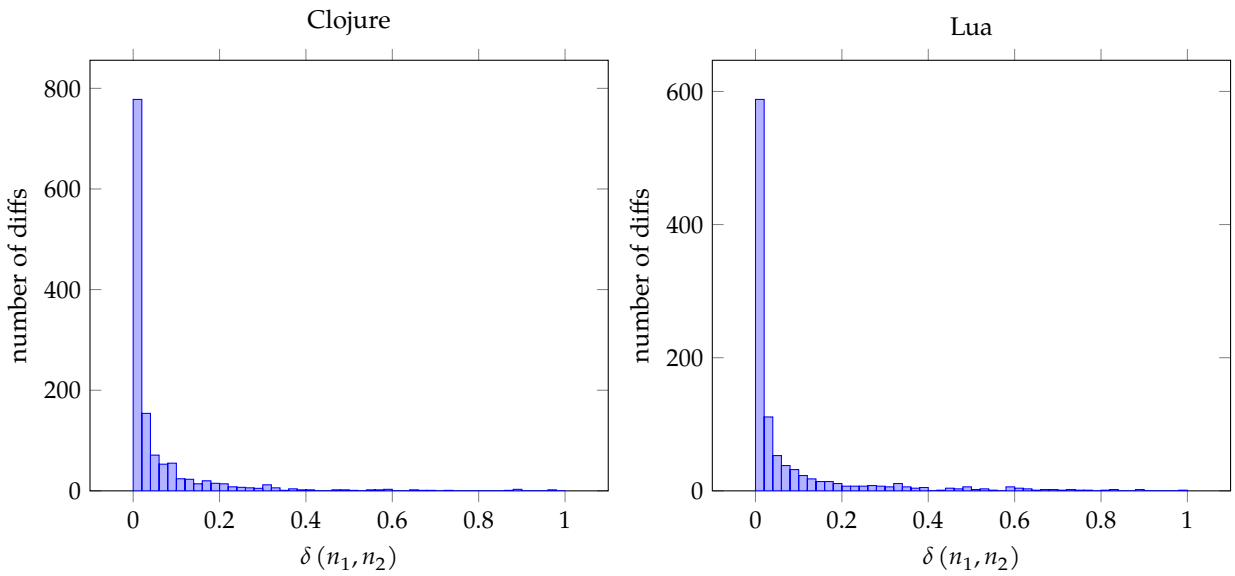


Figure 5.1: Normalized difference between number of nodes in source and target tree, where n_1 and n_2 are the number of nodes in the source and target tree respectively and $\delta(n_1, n_2) = \frac{|n_1 - n_2|}{n_1 + n_2}$

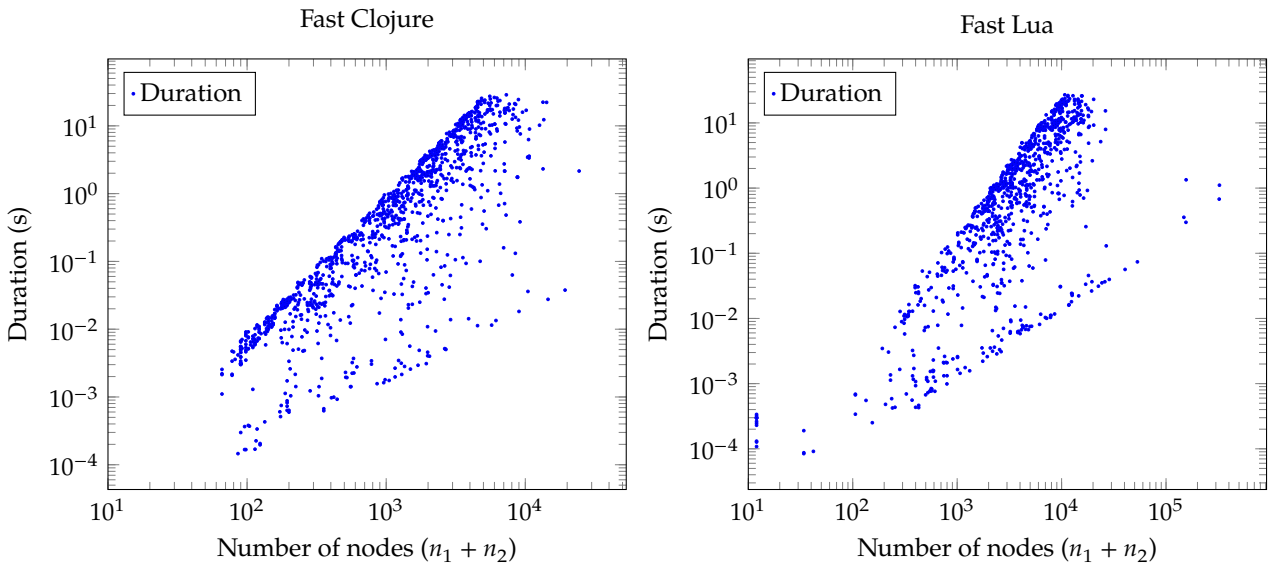


Figure 5.2: Log-log plot of execution time vs node size. A clear quadratic trend is visible

within the specified timeout or ran out of memory. We see that a large percentage of merges did not complete. The algorithm is not only quadratic in time but also in memory, and increasing the timeout any further causes the algorithm to run out of memory on our machines. Furthermore, even if we did not run out of memory, we think that increasing the timeout will not give us many more solved conflicts. Trees that are similar are more likely to produce patches that will merge without conflict as they will generate patches with many copies nodes. Also, trees that are similar are more likely to not hit the wort-case complexity of the algorithm. So the longer the diff algorithm runs, the less likely it is that the produced the *OA* and *OB* patches will actually merge successfully.

Clojure				Lua			
Repository	Total	Merged	Timeout	Repository	Total	Merged	OOM
aleph	45	10	3	Algorithm-Impl	0	0	0
boot	13	0	5	awesome	5	1	0
cascalog	68	5	10	busted	9	0	0
clj-http	15	1	3	CorsixTH	25	3	15
compojure-api	18	2	3	garrysmod	1	0	1
duckling_old	22	2	3	hawkthorne-journey	182	40	25
friend	2	1	0	kong	206	24	18
frontend	33	7	0	koreader	30	3	16
incanter	95	5	15	luakit	43	6	16
kibit	10	3	1	luarocks	46	12	1
lein-figwheel	9	3	1	luci	0	0	0
leiningen	45	6	2	luvit	4	0	0
liberator	20	2	10	minetest_game	0	0	0
Midje	29	9	0	nn	4	0	1
onyx	197	36	49	Penlight	6	3	0
overtone	73	20	10	rnn	8	2	2
pedestal	37	10	2	snabb	134	19	47
quil	10	0	1	tarantool	114	8	58
riemann	7	1	0	telegram-bot	5	1	0
ring	43	6	0	vlsub	0	0	0
Totals	790	118	187	**Totals**	823	105	297

Table 5.2: Amount of solved conflicts per language

6 Conclusion and Future work

6.1 Conclusion

We have created a type-safe structured diffing and merging tool in Haskell. We have shown that structured patches can be extended to work on families of mutually recursive datatypes, allowing us to diff and merge a wide variety of programming languages. Also, we have shown that we can use *gdiff* to drive an algorithm that generates structured patches in $O(n^2)$ time.

We have shown that using this algorithm, we can generate patches of good enough quality that they allow us to automatically resolve many conflicts found in real world Lua and Clojure repositories, even with a rather simply merge strategy. This sets a baseline for the amount of merges a structured merge tool should be able to solve.

However, some issues remain. First, though in theory our tool should work for any programming language for which we have a parser in Haskell, in practice we can only use our tool for languages with relatively small ASTs. This is due to the bug in GHC (Section 4.2.1) which we hope will be fixed in the future. Secondly, the tool does not solve conflicts of files in real-time. In order for a structured merge tool to be adopted by developers, merges should take milliseconds instead of seconds, which is currently not the case. We think there is some interesting future work to explore, that might solve remaining issues that we currently face.

6.2 Future work

Diff between unordered collections

The *gdiff* algorithm only works on ordered trees. However, for many parts of files, ordering of elements might be irrelevant. Think for example of the `build-depends` fields in a Cabal file. Algorithms exist that generalise the greatest common subsequence problem. The assignment problem is a classical linear programming problem that can find an optimal matching between unordered collections. However runtime complexity is even worse than that of the *gdiff* algorithm, which might turn out problematic. It is also not clear to us how one would express a linear programming problem elegantly in Haskell, whilst maintaining the same well-formedness guarantees throughout the implementation of the algorithm like in the *gdiff* implementation.

Better merging algorithm

Using disjointness as a means of merging sets a baseline benchmark, but there are many obvious cases that we currently miss. For example, two trees that are permutations of each other should be trivially mergeable if we have a diff structure that can describe swaps of arguments, but are currently

always considered conflicts in our algorithm. The notion of merges of patches should be studied more formally, so that more conflicts can be solved without the need of domain specific knowledge.

Fast Patches through Merkle trees

Miraldo and Swierstra [24] have designed a new diffing algorithm built on top of Merkle trees. Using the *synthesize* functionality from `generics-mrsop`, we can annotate each node in an AST with a hash of itself and its children. Using this hash, we can very efficiently identify common subtrees between two trees. Using this idea, Miraldo and Swierstra [24] come up with a structured diff that can be calculated in $O(n)$ space and time. The algorithm uses a new patch datatype. This patch datatype does not consist of deletions and insertions throughout the tree, but instead projects out all the places that are not deleted at once, and then assigns these to places in the destination tree. Each projected subtree is assigned a metavariable, where subtrees with the same hash get the same variable assigned. This allows us to describe more rich operations, like contractions and swaps of subtrees. Using this richer patch datatype, more conflicts can potentially be solved automatically. Initial experiments have been run on the same dataset as ours and the results are promising. Patches can be constructed in under 200 milliseconds and the authors are able to solve about the same amount of conflicts as we currently can.

Bibliography

[1] Sven Apel et al. "Semistructured Merge: Rethinking Merge in Revision Control Systems". In: Sept. 2011. DOI: 10.1145/2025113.2025141.

[2] Richard S. Bird and Lambert G. L. T. Meertens. "Nested Datatypes". In: *Proceedings of the Mathematics of Program Construction*. MPC '98. London, UK, UK: Springer-Verlag, 1998, pp. 52–67. ISBN: 3-540-64591-8. URL: http://dl.acm.org/citation.cfm?id=648084.747162.

[3] Joachim Breitner et al. "Safe Zero-cost Coercions for Haskell". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: ACM, 2014, pp. 189–202. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628141. URL: http://doi.acm.org/10.1145/2628136.2628141.

[4] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. "Associated Type Synonyms". In: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ICFP '05. Tallinn, Estonia: ACM, 2005, pp. 241–253. ISBN: 1-59593-064-7. DOI: 10.1145/1086365.1086397. URL: http://doi.acm.org/10.1145/1086365.1086397.

[5] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: ACM, 2000, pp. 268–279. ISBN: 1-58113-202-6. DOI: 10.1145/351240.351266. URL: http://doi.acm.org/10.1145/351240.351266.

[6] Richard A. Eisenberg. "Dependent Types in Haskell: Theory and Practice". In: *CoRR* abs/1610.07978 (2016). arXiv: 1610.07978. URL: http://arxiv.org/abs/1610.07978.

[7] Richard A. Eisenberg. "Dependent Types in Haskell: Theory and Practice". In: *CoRR* abs/1610.07978 (2016). arXiv: 1610.07978. URL: http://arxiv.org/abs/1610.07978.

[8] Richard A. Eisenberg and Stephanie Weirich. "Dependently Typed Programming with Singletons". In: *Proceedings of the 2012 Haskell Symposium*. Haskell '12. Copenhagen, Denmark: ACM, 2012, pp. 117–130. ISBN: 978-1-4503-1574-6. DOI: 10.1145/2364506.2364522. URL: http://doi.acm.org/10.1145/2364506.2364522.

[9] Giovanni Garufi. "Version Control Systems - Diffing with Structure". 2018.

[10] Gérard Huet. "Residual theory in λ-calculus: a formal development". In: *Journal of Functional Programming* 4.3 (1994), pp. 371–394. DOI: 10.1017/S0956796800001106.

[11] Donald E. Knuth. "The Genesis of Attribute Grammars". In: *Proceedings of the International Conference WAGA on Attribute Grammars and Their Applications*. London, UK, UK: Springer-Verlag, 1990, pp. 1–12. ISBN: 3-540-53101-7. URL: http://dl.acm.org/citation.cfm?id=645938.671208.

[12] Ralf Lämmel and Simon Peyton Jones. "Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming". In: *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*. TLDI '03. New Orleans, Louisiana, USA: ACM, 2003, pp. 26–37. ISBN: 1-58113-649-8. DOI: 10.1145/604174.604179. URL: http://doi.acm.org/10.1145/604174.604179.

[13] Eelco Lempsink, Sean Leather, and Andres Löh. "Type-safe Diff for Families of Datatypes". In: *Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming*. WGP '09. Edinburgh, Scotland: ACM, 2009, pp. 61–72. ISBN: 978-1-60558-510-9. DOI: 10.1145/1596614.1596624. URL: http://doi.acm.org/10.1145/1596614.1596624.

[14] Olaf Leßenich, Sven Apel, and Christian Lengauer. "Balancing precision and performance in structured merge". In: *Automated Software Engineering* 22.3 (Sept. 2015), pp. 367–397. DOI: 10.1007/s10515-014-0151-5. URL: https://doi.org/10.1007/s10515-014-0151-5.

[15] Antoni Lozano and Gabriel Valiente. "On the maximum common embedded subtree problem for ordered trees". In: *In C. Iliopoulos and T Lecroq, editors, String Algorithmics, chapter 7. King's College London Publications*. 2004.

[16] José Pedro Magalhães et al. "A Generic Deriving Mechanism for Haskell". In: *Proceedings of the Third ACM Haskell Symposium on Haskell*. Haskell '10. Baltimore, Maryland, USA: ACM, 2010, pp. 37–48. ISBN: 978-1-4503-0252-4. DOI: 10.1145/1863523.1863529. URL: http://doi.acm.org/10.1145/1863523.1863529.

[17] COnor McBride and James McKinna. "The view from the left". In: *Journal of Functional Programming* 14.1 (2004), pp. 69–111. DOI: 10.1017/S0956796803004829.

[18] Erik Meijer, Maarten Fokkinga, and Ross Paterson. "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire". In: *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. Cambridge, Massachusetts, USA: Springer-Verlag New York, Inc., 1991, pp. 124–144. ISBN: 0-387-54396-1. URL: http://dl.acm.org/citation.cfm?id=127960.128035.

[19] T. Mens. "A State-of-the-Art Survey on Software Merging". In: *IEEE Trans. Softw. Eng.* 28.5 (May 2002), pp. 449–462. ISSN: 0098-5589. DOI: 10.1109/TSE.2002.1000449. URL: https://doi.org/10.1109/TSE.2002.1000449.

[20] Victor Cacciari Miraldo. 2018. URL: http://github.com/VictorCMiraldo/stdiff.

[21] Victor Cacciari Miraldo, Pierre-Évariste Dagand, and Wouter Swierstra. "Type-directed Diffing of Structured Data". In: *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Type-Driven Development*. TyDe 2017. Oxford, UK: ACM, 2017, pp. 2–15. ISBN: 978-1-4503-5183-6. DOI: 10.1145/3122975.3122976. URL: http://doi.acm.org/10.1145/3122975.3122976.

[22] Victor Cacciari Miraldo and Alejandro Serrano. "Sums of Products for Mutually Recursive Datatypes". Utrecht, The Netherlands, 2018.

[23] Victor Cacciari Miraldo and Alejandro Serrano. "Sums of Products for Mutually Recursive Datatypes: The Appropriationist's View on Generic Programming". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development*. TyDe 2018. St. Louis, MO, USA: ACM, 2018, pp. 65–77. ISBN: 978-1-4503-5825-5. DOI: 10.1145/3240719.3241786. URL: http://doi.acm.org/10.1145/3240719.3241786.

[24] Victor Cacciari Miraldo and Wouter Swierstra. "An Efficient Algorithm for Type-Directed Structural Diffing". 2019.

[25] Victor Cacciari Miraldo and Wouter Swierstra. *Structure-aware version control A generic approach using Agda*. Tech. rep. UU-CS-2017-002. Utrecht University, 2016.

[26] Thomas van Noort et al. "A Lightweight Approach to Datatype-generic Rewriting". In: *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*. WGP '08. Victoria, BC, Canada: ACM, 2008, pp. 13–24. ISBN: 978-1-60558-060-9. DOI: 10.1145/1411318.1411321. URL: http://doi.acm.org/10.1145/1411318.1411321.

[27] Arian van Putten. 2019. URL: https://github.com/arianvp/generics-mrsop-diff.

[28] Alejandro Serrano and Victor Cacciari Miraldo. "Generic Programming of All Kinds". In: *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*. Haskell 2018. St. Louis, MO, USA: ACM, 2018, pp. 41–54. ISBN: 978-1-4503-5835-4. DOI: 10.1145/3242744.3242745. URL: http://doi.acm.org/10.1145/3242744.3242745.

[29] Tim Sheard and Simon Peyton Jones. "Template Meta-programming for Haskell". In: *SIGPLAN Not.* 37.12 (Dec. 2002), pp. 60–75. ISSN: 0362-1340. DOI: 10.1145/636517.636528. URL: http://doi.acm.org/10.1145/636517.636528.

[30] S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João Saraiva. "Designing and Implementing Combinator Languages". In: *Advanced Functional Programming*. Ed. by S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 150–206. ISBN: 978-3-540-48506-3.

[31] Marco Vassena. "Generic Diff3 for Algebraic Datatypes". In: *Proceedings of the 1st International Workshop on Type-Driven Development*. TyDe 2016. Nara, Japan: ACM, 2016, pp. 62–71. ISBN: 978-1-4503-4435-7. DOI: 10.1145/2976022.2976026. URL: http://doi.acm.org/10.1145/2976022.2976026.

[32] Edsko de Vries and Andres Löh. "True Sums of Products". In: *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*. WGP '14. Gothenburg, Sweden: ACM, 2014, pp. 83–94. ISBN: 978-1-4503-3042-8. DOI: 10.1145/2633628.2633634. URL: http://doi.acm.org/10.1145/2633628.2633634.

[33] Alexey Rodriguez Yakushev et al. "Generic Programming with Fixed Points for Mutually Recursive Datatypes". In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ICFP '09. Edinburgh, Scotland: ACM, 2009, pp. 233–244. ISBN: 978-1-60558-332-7. DOI: 10.1145/1596550.1596585. URL: http://doi.acm.org/10.1145/1596550.1596585.

[34] Brent A. Yorgey et al. "Giving Haskell a Promotion". In: *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. TLDI '12. Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 53–66. ISBN: 978-1-4503-1120-5. DOI: 10.1145/2103786.2103795. URL: http://doi.acm.org/10.1145/2103786.2103795.