

1
2

3
4
5

6
7
8
9
10
11
12
13

**Type-Safe Generic Differencing of
Mutually Recursive Families**

Getypeerde Generieke Differentiatie van Wederzijds
Rekursieve Datatypes
(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht
op gezag van de rector magnificus, prof. dr. G.J. van der Zwaan, in-
gevolge het besluit van het college voor promoties in het openbaar
te verdedigen op October 5, 2020

door

Victor Cacciari Miraldo
geboren op October 16, 1991
te São Paulo, Brasil

14 Promotor: Prof.dr. G. Keller
Copromotor: Dr. W. Swierstra

15 Dit proefschrift werd (mede) mogelijk gemaakt met financiële steun van de
16 Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO),
17 project Revision Control of Structured Data (612.001.401).

18 **CONTENTS**

19	ABSTRACT	v
20	1 INTRODUCTION	1
21	1.1 Contributions and Outline	5
22	2 BACKGROUND	7
23	2.1 Differencing and Edit Distance	8
24	2.1.1 String Edit Distance and UNIX diff	9
25	2.1.2 Classic Tree Edit Distance	12
26	2.1.3 Shortcomings of Edit-Script Based Approaches	14
27	2.1.4 Synchronizing Changes	15
28	2.1.5 Literature Review	18
29	2.2 Generic Programming	20
30	2.2.1 GHC Generics	20
31	2.2.2 Explicit Sums of Products	22
32	2.2.3 Discussion	24
33	3 GENERIC PROGRAMMING WITH MUTUALLY RECURSIVE TYPES	27
34	3.1 The generics-mrsop library	29
35	3.1.1 Explicit Fixpoints with Codes	29
36	3.1.2 Mutual Recursion	33
37	3.1.3 Practical Features	40
38	3.1.4 Example: Well-Typed Classical Tree Differencing	44
39	3.2 The generics-simplistic Library	46
40	3.2.1 The Simplistic View	47
41	3.2.2 Mutual Recursion	48
42	3.2.3 The (Co)Free (Co)Monad	53
43	3.2.4 Practical Features	55
44	3.3 Discussion	58

45	4	STRUCTURAL PATCHES	61
46	4.1	The Type of Patches	63
47	4.1.1	Functorial Patches	63
48	4.1.2	Recursive Changes	67
49	4.2	Merging Patches	70
50	4.3	Computing <i>Patch_{ST}</i>	74
51	4.3.1	Naive enumeration	74
52	4.3.2	Translating from <i>gdiff</i>	76
53	4.4	Discussion	79
54	5	PATTERN-EXPRESSION PATCHES	81
55	5.1	Changes	82
56	5.1.1	A Concrete Example	82
57	5.1.2	Representing Changes Generically	90
58	5.1.3	Meta Theory	92
59	5.1.4	Computing Changes	97
60	5.2	The Type of Patches	102
61	5.2.1	Computing Closures	106
62	5.2.2	The <i>diff</i> Function	109
63	5.2.3	Aligning Changes	109
64	5.2.4	Summary	117
65	5.3	Merging Aligned Patches	118
66	5.4	Discussion and Further Work	132
67	6	EXPERIMENTS	137
68	6.1	Data Collection	138
69	6.2	Performance	138
70	6.3	Synchronization	140
71	6.3.1	Threats to Validity	146
72	6.4	Discussion	147
73	7	DISCUSSION	149

74	7.1 The Future of Structural Differencing	149
75	7.2 Concluding Remarks	151
76	A SOURCE-CODE AND DATASET	153
77	A.1 Source-Code	153
78	A.2 Dataset	153

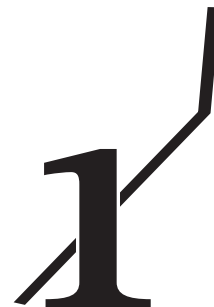
ABSTRACT

80 The UNIX `diff` tool – which computes the differences between two files in terms of a
81 set of copied lines – is widely used in software version control. The fixed *lines-of-code*
82 granularity, however, is sometimes too coarse and obscures simple changes, i.e., renaming
83 a single parameter triggers the whole line to be seen as *changed*. This may lead to
84 unnecessary conflicts when unrelated changes occur on the same line. Consequently, it
85 is difficult to merge such changes automatically.

86 In this thesis we discuss two novel approaches to structural differencing, generically
87 – which work over a large class of datatypes. The first approach defines a type-indexed
88 representation of patches and provides a clear merging algorithm, but it is computation-
89 ally expensive to produce patches with this approach. The second approach addresses
90 the efficiency problem by choosing an extensional representation for patches. This en-
91 ables us to represent transformations involving insertions, deletions, duplication, con-
92 tractions and permutations which are computable in linear time. With the added ex-
93 pressivity, however, comes added complexity. Consequently, the merging algorithm is
94 more intricate and the patches can be harder to reason about.

95 Both of our approaches can be instantiated to mutually recursive datatypes and, con-
96 sequently, can be used to compare elements of most programming languages. Writing
97 the software that does so, however, comes with additional challenges. To address this
98 we have developed two new libraries for generic programming in Haskell.

99 Finally, we empirically evaluate our algorithms by a number of experiments over real
100 conflicts gathered from `GitHub`. Our evaluation reveals that at least 26% of the conflicts
101 that developers face on an everyday basis could have been automatically merged. This
102 suggests there is a benefit in using structural differencing tools as the basis for software
103 version control.



106 INTRODUCTION

107 Version Control is essential for any kind of distributed collaborative work. It enables
108 contributors to operate independently and later combine their work. For that, though,
109 it must address the situation where two developers changed a piece of information in dif-
110 ferent ways. One option is to lock further edits until a human decides how to reconcile
111 the changes, regardless of the changes. Yet, many changes can be reconciled *automati-*
112 *cally*.

113 Software engineers usually rely on version control systems to help with this dis-
114 tributed workflow. These tools keep track of the changes performed to the objects under
115 version control, computing changes between old and new versions of an object. When
116 time comes to reconcile changes, it runs a *merge* algorithm that decides whether the
117 changes can be synchronized or not. At the heart of this process is (A) the representa-
118 tion of changes, usually denoted a *patch* and (B) the computation of a *patch* between
119 two objects.

120 Maintaining software as complex as an operating system with as many as several
121 thousands contributors is a technical feat made possible thanks, in part, to a venerable
122 Unix utility: `UNIX diff` [42]. It computes the line-by-line difference between two tex-
123 tual files, determining the smallest set of insertions and deletions of lines to transform
124 one file into the other. In other words, it tries to share as many lines between source and
125 destination as possible. This is, in fact, the most widespread representation for *patches*,
126 used by tools such as `git`, `mercurial` and `darcs`.

127 The limited grammar of changes used by the `UNIX diff` works particularly well for
128 programming languages that organize a program into lines of code. For example, con-
129 sider the following modification that extends an existing `for`-loop to not only compute
130 the sum of the elements of an array, but also compute their product:

```

131     sum := 0;
132 +   prod := 1;
133     for (i in is) {
134         sum += i;
135 +   prod *= i;
136     }

```

137 However, the bias towards *lines* of code may lead to (unnecessary) conflicts when
 138 considering other programming languages. For instance, consider the following diff be-
 139 tween two Haskell functions that add a new argument to an existing function:

```

140 - head []          = error "?!"
141 - head (x :: xs) = x
142 + head []          d = d
143 + head (x :: xs) d = x

```

144 This modest change impacts all the lines of the function's definition, even though it
 145 affects relatively few elements of the abstract-syntax.

146 The line-based bias of the diff algorithm may lead to unnecessary *conflicts* when
 147 considering changes made by multiple developers. Consider the following innocuous
 148 improvement of the original head function, which improves the error message raised
 149 when the list is empty:

```

150 head []          = error "Expecting a non-empty list."
151 head (x :: xs) = x

```

152 Trying to apply the patch above to this modified version of the head function will fail,
 153 as the lines do not match – even if both changes modify distinct parts of the declaration
 154 in the case of non-empty lists.

155 The inability to identify more fine grained changes in the objects it compares is a
 156 consequence of the *by line* granularity of *patches*. Ideally, however, the objects under
 157 comparison should dictate the granularity of change to be considered. This is precisely
 158 the goal of *structural differencing* tools.

159 If we reconsider the example above, we could give a more detailed description of
 160 the modification made to the head function by describing the changes made to the con-
 161 stituent declarations and expressions:

```

162 head []          {+d+} = error {"?!"-} {"Expect..."+}
163 head (x :: xs) {+d+} = x

```

164 There is more structure here than mere lines of text. In particular, the granularity is at the
 165 abstract-syntax level. It is worthwhile to note that this problem also occurs in languages
 166 which tend to be organized in a line-by-line manner. Modern languages which contain

167 any degree of object-orientation will also group several abstract-syntax elements on the
 168 same line. Take the Java function below,

```
169 public void test(obj) {
170     assert(obj.size(), equalTo(5));
171 }
```

172 Now consider a situation where one developer updated the test to require the size
 173 of `obj` to be 6, but another developer changed the function that makes the comparison,
 174 resulting in the two orthogonal versions below;

```
175 public void test(obj) {                public void test(obj) {
    assert(obj).hasSize(5);                assert(obj.size(), equalTo(6));
    }                                    }
```

176 It is straightforward to see that the desired *synchronized* version can incorporate both
 177 changes, calling `assert(obj).hasSize(6)`. Combining these changes would be impos-
 178 sible without access to information about the old and new state of *individual abstract-*
 179 *syntax elements*. Simple line-based information is insufficient, even in line-oriented lan-
 180 guages.

181 Differencing and synchronization algorithms tend to follow a common framework –
 182 compute the difference between two values of some type *a*, and represent these changes
 183 in some type, *Patch a*. The *diff* function *computes* the differences between two values of
 184 type *a*, whereas *apply* attempts to transform a value according to the information stored
 185 in the *Patch* provided to it.

```
186 diff  :: a → a → Patch a
      apply :: Patch a → a → Maybe a
```

187 A definition of *Patch a* which has access to information about the structure of *a*
 188 enables us to represent changes at a more refined granularity. In Chapters 4 and 5 we
 189 discuss two different definitions of *Patch*, both capturing changes at the granularity of
 190 abstract-syntax elements.

191 Note that the *apply* function is inherently partial, for example, when attempting to
 192 delete data which is not present applying the patch will fail. Yet when it succeeds, the
 193 *apply* function must return a value of type *a*. This may seem like an obvious design
 194 choice, but this property does not hold for the approaches [6, 28] using `xml` or `json`
 195 to represent abstract syntax trees, where the result of applying a patch may produce ill-
 196 typed results, i.e., schema violations.

197 UNIX *diff* [42] follows this very framework too, but for the specific type of lines of
 198 text, taking *a* to be `[String]`. It represents patches as a series of insertions, deletions and
 199 copies of lines and works by enumerating all possible patches that transform the source
 200 into the destination and chooses the best such patch. There have been several attempts
 201 at generalizing these results to handle arbitrary datatypes [107, 24, 82, 51], including

our own attempt discussed in Chapter 4. All of these follow the same recipe: enumerate all combinations of insertions, deletions and copies that transform the source into the destination and choose the ‘best’ one. Consequently, they also suffer from the same drawbacks as classic edit-distance – which include non-uniqueness of the best solution and slow algorithms. We will discuss them in more detail in Section 2.1.1.

Once we have a *diff* and an *apply* functions handy, we move on to the *merge* function, which is responsible for synchronizing two different changes into a single one, when they are compatible. Naturally not all patches can be merged, in fact, we can only merge those patches that alter *disjoint* parts of the AST. Hence, the merge function must be partial, returning a conflict whenever patches change the same part of the tree in different ways.

merge :: *Patch* *a* → *Patch* *a* → *Either Conflicts (Patch a)*

A realistic merge function should naturally distribute conflicts to their specific locations inside the merged patch and still try to synchronize non-conflicting parts of the changes. This is orthogonal to our objective, however. The abstract idea is still the same: two patches can either be reconciled fully or there exists conflicts between them.

The success rate of the *merge* function – that is, how often it is able to reconcile changes – can never be 100%. There will always be changes that require human intervention to be synchronized. Nevertheless, the quality of the synchronization algorithm directly depends on the expressivity of the *Patch* datatype. If *Patch* provides information solely on which lines of the source have changed, there is little we can merge. Hence, we want that values of type *Patch* to carry information about the structure of *a*. Naturally though, we do not want to build domain specific tools for each programming language for which we wish to have source files under version control – which would be at least impractical. The better option is to use a *generic representation*, which can be used to encode arbitrary programming languages, and describe the *Patch* datatype generically.

Structural differencing is a good example of the need for generic programming: we would like to have differencing algorithms to work over arbitrary datatypes, but maintaining the type-safety that a language like Haskell provides. This added safety means that all the manipulations we perform on the patches are guaranteed to never produce ill-formed elements, which is a clear advantage over using something like XML to represent our data, even though there exists differencing tools that use XML as their underlying representation for data. We refer to these as *untyped* tree differencing algorithms in contrast the *typed* approach, which guarantees type safety by construction.

The Haskell type-system is expressive enough to enable one to write *typed* generic programming algorithms. These algorithms, however, can only be applied to datatypes that belong in the set of types handled by the generic programming library of choice. For example, the *regular* [76] approach is capable of handling types which have a *regular* recursive structure – lists, *n*-ary trees, etc –, but cannot represent nested types, for

example. In Section 2.2 we will give an overview of existing approaches to generic programming in Haskell. No library, however, was capable of handling mutually recursive types – which is the universe of datatypes that context free languages belong in – in a satisfactory manner. This means that to explore differencing algorithms for various programming languages we would have to first develop the generic programming functionality necessary for it. Gladly, Haskell’s type system has evolved enough since the initial efforts on generic programming for mutually recursive types (`multirec` [105]), enabling us to write significantly better libraries, as we will discuss in Chapter 3.

1.1 CONTRIBUTIONS AND OUTLINE

This thesis documents a number of peer-reviewed contributions, namely:

- a) Chapter 3 discusses the `generics-mrsop` [70] library, which offers combinator-based generic programming for mutually recursive families. This work came out of close collaboration with Alejandro Serrano on a variety of generic programming topics.
- b) Chapter 4 is derived from a paper published with [69] with Pierre-Évariste Dagand. We worked closely together to define a type-indexed datatype used to represent changes in a more structured way than edit-scripts. Chapter 4 goes further into developing a merging algorithm and exploring different ways to compute patches given two concrete values.
- c) Chapter 5 is the refinement of our paper [71] on an efficient algorithm for computing patches, where we tackle the problems from Chapter 4 with a different representation for patches altogether.

Other contributions that have not been peer-reviewed include:

- d) Chapter 3 discusses the `generics-simplistic` library, a different approach to generic programming that overcomes an important space leak in the Haskell compiler, which rendered `generics-mrsop` unusable in large, real-world, examples.
- e) Chapter 5 introduces a merging algorithm and Chapter 6 discusses its empirical evaluation over a dataset of real conflicts extracted from GitHub.

270



271

272 BACKGROUND

273 The most popular tool for computing differences between two files is the UNIX `diff` [42],
274 it works by comparing files in a *line-by-line* basis and attempts to match lines from the
275 source file to lines in the destination file. For example, consider the two files below:

276	1	res := 0;	1	print("summing up");
	2	for (i in is) {	2	sum := 0;
	3	res += i;	3	for (i in is) {
	4	}	4	sum += i;
			5	}

277 Lines 2 and 4 in the source file, on the left, match lines 3 and 5 in the destination.
278 These are identified as copies. The rest of the lines, with no matches, are marked as
279 deletions or insertions. In this example, lines 1 and 3 in the source are deleted and lines
280 1,2 and 4 in the destination are inserted.

281 This information about which lines have been *copied*, *deleted* or *inserted* is then pack-
282 aged into an *edit-script*: a list of operations that transforms the source file into the desti-
283 nation file. For the example above, the edit-script would read something like: delete the
284 first line; insert two new lines; copy a line; delete a line; insert a line and finally copy the
285 last line. The output we would see from the UNIX `diff` would show deletions prefixed
286 with a minus sign and insertions prefixed with a plus sign. Copies have no prefix. In our
287 case, it would look something like:

```
288 - res := 0;
289 + print("summing up");
290 + sum := 0;
291   for (i in is) {
292 -   res += i;
293 +   sum += i;
294 }
```

The edit-scripts produced by the UNIX `diff` contain information about transforming the source into the destination file by operating exclusively at the *lines-of-code* level. Computing and representing differences in a finer granularity than *lines-of-code* is usually done by parsing the data into a tree and later flattening said tree into a list of nodes, where one then reuses existing techniques for computing differences over lists, i.e., think of printing each constructor of the tree into its own line. This is precisely how most of the classic work on tree edit distance computes tree differences (Section 2.1.2).

Recycling linear edit distance into tree edit distance, however, also comes with its drawbacks. Linear differencing uses *edit-scripts* to represent the differences between two objects. Edit-scripts are composed of atomic operations, which traditionally include operations such as *insert*, *delete* and *copy*. These scripts are later interpreted by the application function, which gives the semantics to these operations. The notion of *edit distance* between two objects is defined as the cost of the least cost *edit-script* between them, where cost is some defined metric, often context dependent. One major drawback, for example, is the least cost edit-script is chosen arbitrarily in some situations, namely, when it is not unique. This makes the results computed by these algorithms hard to predict. Another issue, perhaps even more central, are the algorithms that arise from this ambiguity which are inherently slow.

The algorithms computing edit-scripts must either return an approximation of the least cost edit-script or check countless ambiguous choices to return the optimal one. Finally, manipulating edit-scripts in an untyped fashion, say, for instance in order to merge them, might produce ill-typed trees – as in *not abiding by a schema* – as a result [100]. We can get around this last issue by writing edit-scripts in a typed form [51], but this requires some non-trivial generic programming techniques to scale.

The second half of this chapter is the state-of-the-art of the generic programming ecosystem in Haskell. Including the `GHC.Generics` and `generics-sop` libraries, which introduce all the necessary parts for us to build our own solutions later, in Chapter 3.

2.1 DIFFERENCING AND EDIT DISTANCE

The *edit distance* between two objects is defined as the cost of the least-cost edit-script that transforms the source object into the target object – that is, the edit-script with the least insertions and deletions. Computing edit-scripts is often referred to as *differencing* objects. Where edit distance computation is only concerned with how *similar* one object is to another, *differencing*, on the other hand, is actually concerned with how to transform one objects into another. Although very closely related, these do make up different problems. In the biology domain [2, 39, 62], for example, one is concerned solely in finding similar structures in a large set of structures, whereas in software version control systems manipulating and combining differences is important.

332 The wide applicability of differencing and edit distances leads to a variety of cost
 333 notions, edit-script operations and algorithms for computing them [15, 13, 79]. In this
 334 section we will review some of the important notions and background work on edit dis-
 335 tance. We start by looking at the string edit distance (Section 2.1.1) and then generalize
 336 this to untyped trees (Section 2.1.2), as it is classically portrayed in the literature, which
 337 is reviewed in Section 2.1.5. Finally, we discuss some of the consequences of working
 338 with typed trees in Section 3.1.4.

339 2.1.1 STRING EDIT DISTANCE AND UNIX `DIFF`

340 In this section we look at two popular notions of edit distance. The *Levenshtein Dis-*
 341 *tance* [52, 13], for example, works well for detecting spelling mistakes[74] or measuring
 342 how similar two languages are [96]. It considers insertions, deletions and substitutions
 343 of characters as its edit operations. The *Longest Common Subsequence (LCS)* [13], on
 344 the other hand, considers insertions, deletions and copies as edit operations and is bet-
 345 ter suited for identifying shared sequences between strings.

346 **LEVENSHTEIN DISTANCE** The Levenshtein distance regards insertions, deletions and
 347 substitutions of characters as edit operations, which can be modeled in Haskell by the
 348 *EditOp* datatype below. Each of those operations have a predefined *cost* metric.

```

data EditOp = Ins Char | Del Char | Subst Char Char
cost :: EditOp → Int
349 cost (Ins _)      = 1
    cost (Del _)    = 1
    cost (Subst c d) = if c == d then 0 else 1

```

350 These individual operations are then grouped into a list, usually denoted an *edit-*
 351 *script*. The *apply* function, below, gives edit-scripts a denotational semantics by mapping
 352 them to partial functions over *Strings*.

```

apply :: [EditOp] → String → Maybe String
apply [] [] = Just []
353 apply (Ins c : ops) ss = (c :) <$> apply ops ss
    apply (Del c : ops) (s : ss) = guard (c == s) > apply ops ss
    apply (Subst c d : ops) (s : ss) = guard (c == s) > (d :) <$> apply ops ss

```

354 The *cost* metric associated with these edit operations is defined to force substitutions
 355 to cost less than insertions and deletions. This ensures that the algorithm looking for the
 356 list of edit operations with the minimum cost will prefer substitutions over deletions and
 357 insertions.

```

lev :: String → String → [EditOp]
lev [] [] = []
lev (x : xs) [] = Del x : lev xs []
lev [] (y : ys) = Ins y : lev [] ys
lev (x : xs) (y : ys) = let i = Ins y : lev (x : xs) ys
                        d = Del x : lev xs (y : ys)
                        s = Subst x y : lev xs ys
                        in minimumBy cost [i, d, s]

```

FIGURE 2.1: Definition of the function that returns the edit-script with the minimum Levenshtein Distance between two strings.

We can compute the *edit-script*, i.e. a list of edit operations, with the minimum cost quite easily with a brute-force and inefficient implementation. Figure 2.1 shows the implementation of the edit-script with the minimum Levenshtein distance.

```

levenshteinDist :: String → String → Int
levenshteinDist s d = cost (head (lev s d))

```

Note that although the Levenshtein distance is unique, the edit-scripts witnessing it is *not*. Consider the case of lev “ab” “ba” for instance. All of the edit-scripts below have cost 2, which is the minimum possible cost.

```

lev “ab” “ba” ∈ [[Del 'a', Subst 'b' 'b', Ins 'a']
, [Ins 'b', Subst 'a' 'a', Del 'b']
, [Subst 'a' 'b', Subst 'b' 'a']]

```

From a edit distance point of view, there is not an issue. The Levenshtein distance between “ab” and “ba” is 2, regardless of the edit-script. But from an operational point of view, i.e., transforming one string into another, this ambiguity poses a problem. The lack of criteria to favor one edit-script over another means that the result of the differencing algorithm is hard to predict. Consequently, developing a predictable diff and merging algorithm becomes a difficult task.

LONGEST COMMON SUBSEQUENCE

Given our context of source-code version-control, we are rather interested in the *Longest Common Subsequence (LCS)*, which is a restriction of the Levenshtein distance and forms the specification of the UNIX `diff` [42] utility.

```

lcs :: [String] → [String] → [EditOp]
lcs [] [] = []
lcs (x : xs) [] = Del x : lcs xs []
lcs [] (y : ys) = Ins y : lcs [] ys
lcs (x : xs) (y : ys) = let i = Ins y : lcs (x : xs) ys
                        d = Del x : lcs xs (y : ys)
                        s = if x == y then [Cpy : lcs xs ys] else []
                        in minimumBy cost (s # [i, d])

```

FIGURE 2.2: Specification of the UNIX *diff*.

376 If we take the *lev* function and modify it in such a way that it only considers identity
 377 substitutions, that is, *Subst* $x\ y$ with $x \equiv y$, we end up with a function that computes
 378 the classic longest common subsequence. Note that this is different from the longest
 379 common substring problem, as subsequences need not be contiguous.

380 UNIX *diff* [42] performs a slight generalization of the LCS problem by considering
 381 the distance between two *files*, seen as a list of *strings*, opposed to a list of *characters*.
 382 Hence, the edit operations become:

```

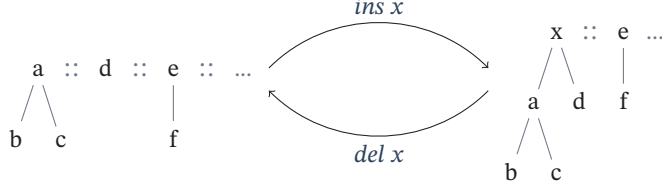
data EditOp = Ins String | Del String | Cpy
cost :: EditOp → Int
383 cost (Ins _) = 1
cost (Del _) = 1
cost Cpy = 0

```

384 The application function is analogous to the *apply* for the Levenshtein distance. The
 385 computation of the minimum cost edit-script, however, is not. We must ensure to issue
 386 a *Cpy* only when both elements are the same, as illustrated in Figure 2.2.

387 Running the *lcs* $x\ y$ function, Figure 2.2, will yield an *edit-script* that enables us to
 388 read out one longest common subsequence of x and y . Note that the ambiguity problem is
 389 still present, however to a lesser degree than with the Levenshtein distance. For example,
 390 there are only two edit-scripts with minimum cost on *lcs* [“a”, “b”] [“b”, “a”]. This, in
 391 fact, is a general problem with any *edit-script* based approaches.

392 The UNIX *diff* implementation uses a number of algorithmic techniques that make
 393 it performant. First, it is essential to use a memoized *lcs* function to avoid recomputing
 394 sub-problems. It is also common to hash the data being compared to have amortized
 395 constant time comparison. More complicated, however, is the adoption of a number of
 396 heuristics that tend to perform well in practice. One example is the *diff* *--patience*
 397 algorithm [17], that will emphasize the matching of lines that appear only once in the
 398 source and destination files.

FIGURE 2.3: Insertion and Deletion of node x , with arity 2 on a forest

```

data EOp = Ins Label | Del Label | Cpy
data Tree = Node Label [Tree]
arity :: Label → Int
apply :: [EOp] → [Tree] → Maybe [Tree]
apply [] [] = Just []
apply (Cpy : ops) ts = apply (Ins l : Del l : ops) ts
apply (Del l : ops) (Node l' xs : ts) = guard (l ≡ l') > apply ops (xs ++ ts)
apply (Ins l : ops) ts
  = (λ(args, rs) → Node l args : rs) ∘ takeDrop (arity l) <$> apply ops ts

```

FIGURE 2.4: Definition of *apply* for tree edit operations

399 2.1.2 CLASSIC TREE EDIT DISTANCE

400 UNIX `diff` can be generalized to compute an edit-script between lists containing data
 401 of arbitrary types. The only requirement being that we must be able to compare this data
 402 for equality. Generalizing over the shape of the data – trees instead of lists – gives rise
 403 to the notion of (untyped) tree edit distance [3, 24, 46, 15, 9, 20]. It considers *arbitrary*
 404 trees as the objects under scrutiny. This added degree of freedom carries over to the
 405 choice of edit operations. Suddenly, there are more edit operations one could use to
 406 create edit-scripts. To name a few, we can have flattening insertions and deletions, where
 407 the children of the deleted node are inserted or removed in-place in the parent node.
 408 Another operation that only exists in the untyped world is node relabeling. This degree
 409 of variation is responsible for the high number of different approaches and techniques
 410 we see in practice [29, 38, 28, 79, 31], Section 2.1.5.

411 Basic tree edit distance [24], however, considers only node insertions, deletions and
 412 copies. The cost function is borrowed entirely from string edit distance together with the
 413 longest common subsequence function, that instead of working with `[a]` will now work
 414 with `[Tree]`. Figure 2.3 illustrates insertions and deletions of (untyped) labels on a forest.
 415 The interpretation of these edit operations as actions on forests is shown in Figure 2.4.

We label these approaches as *untyped* because there exists edit-scripts that yield non-well formed trees. For example, imagine l is a label with arity 2 – supposed to receive two arguments. Now consider the edit-script $\text{Ins } l : []$, which will yield the tree $\text{Node } l []$ once applied to the empty forest. If the objects under differencing are required to abide by a certain schema, such as abstract syntax trees for example, this becomes an issue. This is particularly important when one wants the ability to manipulate patches independently of the objects they have been created from. Imagine a merge function that needs to construct a patch based on two other patches. A wrong implementation of said merge function can yield invalid trees for some given schema. In the context of abstract-syntax, this could be unparseable programs.

It is possible to use the Haskell type system to our advantage and write *EOP* in a way that it is guaranteed to return well-typed results. Labels will be the different constructors of the family of types in question and their arity comes from how many fields each constructor expects. edit-scripts will then be indexes by two lists of types: the types of the trees it consumes and the types of the trees it produces. We will come back to this in more detail in Section 3.1.4, where we review the approach of Lempink and Löh [51] at adapting this untyped framework to be type-safe by construction.

Although edit-scripts provide a very intuitive notion of local transformations over a tree, there are many different edit-scripts that perform the same transformation: the order of insertions and deletions do no matter. This makes it hard to develop algorithms based solely on edit-scripts. The notion of *tree mapping* often comes in handy. It works as a *normal form* version of edit-scripts and represents only the nodes that are either relabeled or copied. We must impose a series of restrictions on these mappings to maintain the ability to produce edit-scripts out of it. Figure 2.5 illustrates four invalid and one valid such mappings.

Definition 2.1.1 (Tree Mapping). Let t and u be two trees, a tree mapping between t and u is an order preserving partial bijection between the nodes of a flattened representation of t and u according to their preorder traversal. Moreover, it preserves the ancestral order of nodes. That is, given two subtrees x and y in the domain of the mapping m , then x is an ancestor of y if and only if $m\ x$ is an ancestor of $m\ y$.

The tree mapping determines the nodes where either a copy or substitution must be performed. Everything else must be deleted or inserted and the order of deletions and insertions is irrelevant, which removes the redundancy of edit-scripts. Nevertheless, the definition of tree mapping is still very restrictive: (i) the “bijective mapping” does not enable trees to be duplicated or contracted; (ii) the “order preserving” does not enable trees to be permuted or moved across ancestor boundaries. These restrictions are there to ensure that one can always compute an edit-script from a tree mapping.

Most tree differencing algorithms start by producing a tree mapping and then extracting an edit-script from this. There are a plethora of design decisions on how to produce a mapping and often the domain of application of the tool will enable one to

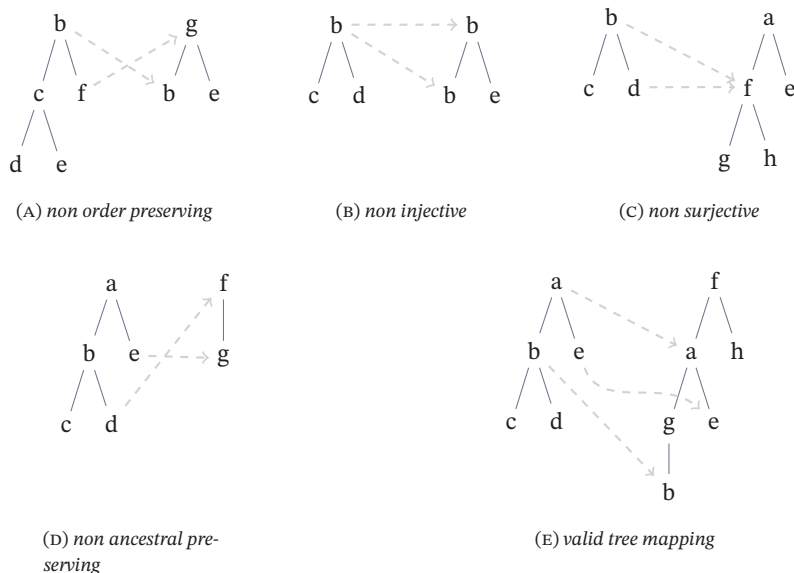


FIGURE 2.5: A number of invalid tree mappings with one valid example.

impose extra restrictions to attempt to squeeze maximum performance out of the algorithm. The `LaDiff` [21] tool, for example, works for hierarchically structured trees – used primarily for \LaTeX source files – and uses a variant of the LCS to compute matchings of elements appearing in the same order, starting at the leaves of the document. Tools such as `XyDiff` [22], used to identify changes in XML documents, use hashes to produce matchings efficiently.

2.1.3 SHORTCOMINGS OF EDIT-SCRIPT BASED APPROACHES

We argue that regardless of the process by which an edit-script is obtained, edit-scripts have inherent shortcomings when they are used to compare tree structured data. The first and most striking is that the use of heuristics to compute optimal solutions is unavoidable. Consider the tree-edit-scripts between the following two trees:



From an *edit distance* point of view, their distance is 2. This fact can be witnessed by two distinct edit-scripts: either [*Cpy Bin* , *Del T* , *Cpy U* , *Ins T*] or [*Cpy Bin* , *Ins U* , *Cpy T* , *Del U*] transform the target into the destination correctly. Yet, from a *differencing* point of view, these two edit-scripts are fairly different. Do we care more about *U* or *T*? What if *U* and *T* are also trees, but happen to have the same size (so that inserting one or the other yields edit-scripts with equal costs)? Ultimately, differencing algorithms that support no *swap* operation must choose to copy *T* or *U* arbitrarily. This decision is often guided by heuristics, which makes the result of different algorithms hard to predict. Moreover, the existence of this type of choice point inherently slows algorithms down since the algorithm *must decide* which tree to copy.

Another issue when dealing with edit-script is that they are type unsafe. It is quite easy to write an edit-script that produces an *ill-formed* tree, according to some arbitrary schema. Even when writing the edit operations in a type-safe way [51] the synchronization of said changes is not guaranteed to be type-safe [100].

Finally, we must mention the lack of expressivity that comes from edit-scripts, from the *differencing* point of view. Consider the trees below,



Optimal edit-scripts oblige us to chose between copying *A* as the left or the right subtree. There is no possibility to represent duplications, permutations or contractions of subtrees. This means that a number of common changes, such as refactorings, yield edit-scripts with a very high cost even though a good part of the information being deleted or inserted should really have been copied.

2.1.4 SYNCHRONIZING CHANGES

When managing local copies of replicated data such as in software version control systems, one is inevitably faced with the problem of *synchronizing* [11] or *merging* changes – when an offline machine goes online with new versions, when two changes happened simultaneously, etc. The *synchronizer* is responsible to identify what has changed and reconcile these changes when possible. Most modern synchronizers operate over the diverging replicas and last common version, without knowledge of the history of the last common version – these are often denoted *state-based* synchronizers, as opposed to *operation-based* synchronizers, which access the whole history of modifications.

The *diff3* [87] tool, for example, is the most widely used synchronizer for textual data. It is a *state-based* that calls the UNIX *diff* to compute the differences between the common ancestor and each diverging replica, then tries to produce an edit-script

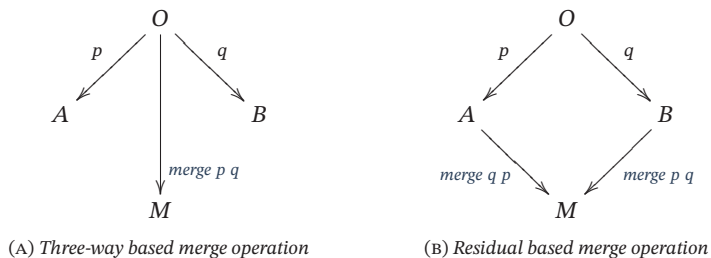


FIGURE 2.6: Two different ways to look at the merge problem.

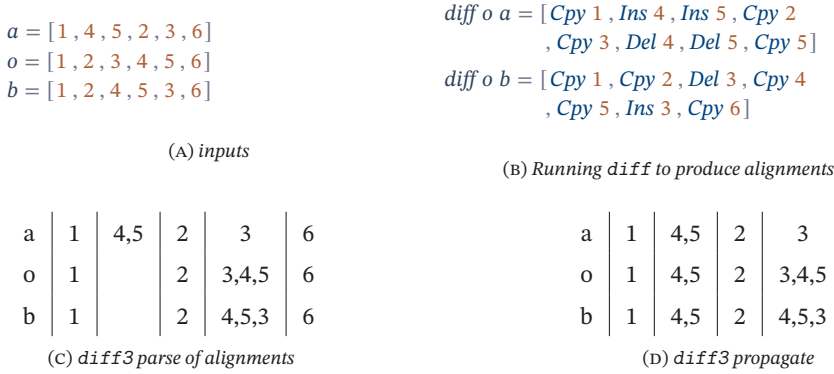
<pre> sum := 0; for (i in is) { sum := sum + i; } </pre> <p>(A) Replica A</p>	<pre> res := 0; for (i in is) { res := res + i; } </pre> <p>(B) Common ancestor, O</p>	<pre> res := 0; sum := 0; for (i in is) { res := res + i; sum := sum + i; } </pre> <p>(C) Replica B</p>
<pre> - res := 0; + sum := 0; for (i in is) { - res := res + i; + sum := sum + i; } </pre> <p>(D) diff O A</p>	<pre> res := 0; + prod := 1; for (i in is) { res := res + i; + prod := prod * i; } </pre> <p>(E) diff O B</p>	

FIGURE 2.7: Two UNIX *diff* patches that diverge from a common ancestor.

502 that when applied to the common ancestor produces a new file, containing the union
 503 of changes introduced in each individual replica. The algorithm itself has been studied
 504 formally [44] and there are proposals to extend it to tree-shaped data [53, 100].

505 Generally speaking, synchronization of changes p and q can be modeled in one of
 506 two ways. Either we produce one change that works on the common ancestor of p and
 507 q , as in Figure 2.6(A), or we produce two changes that act directly on the images of p
 508 and q , Figure 2.6(B). We often call the former a *three-way merge* and the later a *residual*
 509 *merge*.

510 Residual merges, specially if based on actual residual systems [14] pose a few tech-
 511 nical challenges — proving the that the laws required for establishing an actual residual
 512 system is non-trivial. Moreover, they tend to be harder to generalize to n -ary inputs.
 513 They do have the advantage of enabling one to model merges as pushouts [66], which
 514 could provide a desirable metatheoretical foundation.

FIGURE 2.8: A simple *diff3* run

Regardless of whether we choose a *three-way* or *residual* based approach, any state-based synchronizer will invariably have to deal with the problem of *aligning* the changes. That is, deciding which parts of the replicas are copies from the same piece of information in the common ancestor. For example, successfully synchronizing the replicas in Figure 2.7 depends in recognizing that the insertion of `prod := 1`; comes after modifying `res := 0`; to `sum := 0`;. This fact only becomes evident after we look at the result of calling the UNIX `diff` on each diverging replica – the copies in each patch identify which parts of the replicas are ‘the same’.

Figure 2.8 illustrates a run of *diff3* in a simple example, borrowed from Khanna et al. [44], where Alice swaps 2, 3 for 4, 5 in the original file but Bob moves 3 before 6. In a very simplified way, the first thing that happens if we run *diff3* in the inputs (Figure 2.8(A)) is that *diff3* will compute the longest common subsequences between the objects, essentially yielding the alignments it needs (Figure 2.8(B)). The next step is to put the copies side by side and understand which regions are *stable* or *unstable*. The stable regions are those where no replicas changed. In our case, its on 1, 2 and 6 (Figure 2.8(C)). Finally, *diff3* can decide which changes to propagate and which changes are a conflict. In our case, the 4, 5 was only changed in one replica, so it is safe to propagate (Figure 2.8(D)).

Different synchronization algorithms will naturally offer slightly different properties, yet, one that seems to be central to synchronization is locality [44] – which is enjoyed by *diff3* [44]. Locality states that changes to distinct locations of a given object can always be synchronized without conflicts. In fact, we argue this is the only property we can expect out of a general purpose generic synchronizer. The reason being that said synchronizer can rely solely on propositional equality of trees and structural disjointness as the criteria to establish changes as synchronizable. Other criteria will invariantly re-

quire knowledge of the semantics of the data under synchronization. It is worth noting that although “distinct locations” is difficult to define for an underlying list, tree shaped data has the advantage of possessing simpler such notions.

2.1.5 LITERATURE REVIEW

With some basic knowledge of differencing and edit-distances under our belt, we briefly look over some of the relevant literature on the topic. Zhang and Sasha [107] were perhaps the first to provide a number of algorithms which were later improved on by Klein et al. [46] and Dulucq et al. [25]. Finally, Demaine et al. [24] presents an algorithm of cubic complexity and proves this is the best possible worst case. Zhang and Sasha’s algorithm is still preferred in many practical scenarios, though. The more recent *RTED* [82] algorithm maintains the cubic worst case complexity and compares or outruns any of the other algorithms, rendering it the standard choice for computing tree edit distance based on the classic edit operations. In the case of unordered trees the best we can rely on are approximations [7, 8] since the problem is NP-hard [108].

Tree edit distance has seen multidisciplinary interest. From Computational Biology, where it is used to align phylogentic trees and compare RNA secondary structures [2, 39, 62], all the way to intelligent tutoring systems where we must provide good hints to students’ solutions to exercises by understanding how far they are from the correct solutions [80, 91]. In fact, from the *tree edit distance* point of view, we are only concerned with a number, the *distance* between objects, quantifying how similar they are.

From the perspective of *tree differencing*, on the other hand, we actually care about the edit operations and might want to perform computations such as composition and merging of differences. Naturally, however, the choice of edit operations heavily influences the complexity of the *diff* algorithm. Allowing a *move* operation already renders string differencing NP-complete [94]. Tree differencing algorithms, therefore, tend to run approximations of the best edit distance. Most of them still suffer from at least quadratic time complexity, which is prohibitive for most practical applications or are defined for domain specific data, such as the `latexdiff` [97] tool. A number of algorithms specific for XML and imposing different requirements on the schemas have been developed [83]. `LaDiff` [21], for example, imposes restrictions on the hierarchy between labels, it is implemented into the `DiffXML` [73] and `GumTree` [28] tools and is responsible from deducing an edit-script given tree matchings, the tree matching phase differs in each tool. A notable mention is the `XyDiff` [22], which uses hashes to compute matchings and, therefore, supports *move* operations maintaining almost linear complexity. This is perhaps the closest to our approach in Chapter 5. The `RWS-Diff` [31] uses approximate matchings by finding trees that are not necessarily equal but *similar*. This yields a robust algorithm, which is applicable in practice. Most of these techniques recycle list differencing and can be seen as some form of string differencing over the pre-order (or postorder) traversal of trees, which has quadratic upper bound [37]. A careful

579 encoding of the edit operations enables one to have edit-scripts that are guaranteed to
 580 preserve the schema of the data under manipulation [51].

581 When it comes to synchronization of changes [11], the algorithms are heavily depen-
 582 dent on the representation of objects and edit-scripts imposed by the underlying differ-
 583 encing algorithm. The `diff3` [87] tool, developed by Randy Smith in 1988, is still the
 584 most widely used synchronizer. It has received a formal treatment and specification [44]
 585 posterior to its development. Algorithms for synchronizing changes over tree shaped
 586 data include 3DM [53] which merges changes over XML documents, Harmony [32], which
 587 works internally with unordered edge-labelled trees and is focused primarily on un-
 588 ordered containers and, finally, FCDP [50], which uses XML as its internal represen-
 589 tation.

590 Also worth mentioning is the generalization of `diff3` to tree structured data using
 591 well-typed approaches due to Vassena [100], which shows that typed edit-scripts might
 592 not be the best underlying framework for this, as one needs to manually type-check the
 593 resulting edit-scripts.

594 Besides source-code differencing there is patch inference and generation tools. Some
 595 infer patches from human created data [45], whereas other, such as Coccinelle [5, 81],
 596 receive as input a number of diffs, P_0, \dots, P_n , that come from differencing many source
 597 and target files, $P_i = \text{diffs}_i t_i$. The objective then is to infer a common transformation
 598 that was applied everywhere. One can think of determining the *common denominator*
 599 of P_0, \dots, P_n . Refactoring and Rewriting Tools [63, 57] must also be mentioned. Some of
 600 these tools define each supported language AST separately [19, 47], whereas others [99]
 601 support a universal approach similar to *S-expressions*. They identify only parenthesis,
 602 braces and brackets and hence, can be applied to a plethora of programming languages
 603 out-of-the-box.

2.2 GENERIC PROGRAMMING

We would like to consider richer datatypes than *lines-of-text*, without having to define separate *diff* functions for each of them. (*Datatype*-)generic programming provides a mechanism for writing functions by induction on the *structure* of algebraic datatypes [35]. A widely used example is the **deriving** mechanism in Haskell, which frees the programmer from writing repetitive functions such as equality [58]. A vast range of approaches are available as preprocessors, language extensions, or libraries for Haskell [90, 56].

The core idea behind generic programming is the fact that a number of datatypes can be described in a uniform fashion. Hence, if a programmer were to write programs that work over this uniform representation, these programs would immediately work over a variety of datatypes. In this section we look into two modern approaches to generic programming which are widely used, then discuss their design space and drawbacks.

2.2.1 GHC GENERICS

The `GHC.Generics` [55] library, which comes bundled with GHC since version 7.2 and defines the representation of datatypes in terms of uniform *pattern functors*. Consider the following datatype of binary trees with data stored in their leaves:

```
data Bin a = Leaf a | Bin (Bin a) (Bin a)
```

A value of type `Bin a` consists of a choice between two constructors. For the first choice, it also contains a value of type `a` whereas for the second it contains two subtrees as children. This means that the `Bin a` type is isomorphic to `Either a (Bin a, Bin a)`. Different libraries differ on how they define their underlying representations. The representation of `Bin a` in terms of *pattern functors* is written as:

```
Rep (Bin a) = K1 R a :+: (K1 R (Bin a) :*: K1 R (Bin a))
```

The `Rep (Bin a)` above is a direct translation of `Either a (Bin a, Bin a)`, but using the combinators provided by `GHC.Generics`. In addition, we also have two conversion functions `from :: a → Rep a` and `to :: Rep a → a` which form an isomorphism between `Bin a` and `Rep (Bin a)`. The interface ties everything under a typeclass:

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

$$\begin{aligned}
& \text{size } (\text{Bin } (\text{Leaf } 1) (\text{Leaf } 2)) \\
&= \text{gsize } (\text{from}_{\text{gen}} (\text{Bin } (\text{Leaf } 1) (\text{Leaf } 2))) \\
&= \text{gsize } (R1 \ (K1 \ (\text{Leaf } 1) \text{ :*: } K1 \ (\text{Leaf } 2))) \\
&= \text{gsize } (K1 \ (\text{Leaf } 1)) + \text{gsize } (K1 \ (\text{Leaf } 2)) \\
&\stackrel{\dagger}{=} \text{size } (\text{Leaf } 1) + \text{size } (\text{Leaf } 2) \\
&= \text{gsize } (\text{from}_{\text{gen}} (\text{Leaf } 1)) + \text{gsize } (\text{from}_{\text{gen}} (\text{Leaf } 2)) \\
&= \text{gsize } (L1 \ (K1 \ 1)) + \text{gsize } (L1 \ (K1 \ 2)) \\
&= \text{size } (1 :: \text{Int}) + \text{size } (2 :: \text{Int})
\end{aligned}$$
FIGURE 2.9: Reduction of $\text{size } (\text{Bin } (\text{Leaf } 1) (\text{Leaf } 2))$

632 Defining a generic function is done in two steps. First, we define a class that exposes
 633 the function for arbitrary types, in our case, *size*, which we implement for any type via
 634 *gsize*:

```

class Size (a :: *) where
  size :: a → Int
635 instance (Size a) ⇒ Size (Bin a) where
  size = gsize ∘ from_gen

```

636 Next we define the *gsize* function that operates on the level of the representation
 637 of datatypes. We have to use another class and the instance mechanism to encode a
 638 definition by induction on representations:

```

class GSize (rep :: * → *) where
  gsize :: rep x → Int
instance (GSize f, GSize g) ⇒ GSize (f :*: g) where
639   gsize (f :*: g) = gsize f + gsize g
instance (GSize f, GSize g) ⇒ GSize (f :+: g) where
  gsize (L1 f) = gsize f
  gsize (R1 g) = gsize g

```

640 We still have to handle the cases where we might have an arbitrary type in a position,
 641 modeled by the constant functor *K1*. We require an instance of *Size* so we can success-
 642 fully tie the recursive knot.

```

instance (Size x) ⇒ GSize (K1 R x) where
643   gsize (K1 x) = size x

```

To finish the description of the generic *size*, we also need instances for the *unit*, *void* and *metadata* pattern functors, called *U1*, *V1*, and *M1* respectively. Their *GSize* is rather uninteresting, so we omit them for the sake of conciseness.

This technique of *mutually recursive classes* is quite specific to the `GHC.Generics` flavor of generic programming. Figure 2.9 illustrates how the compiler goes about choosing instances for computing *size* (*Bin* (*Leaf* 1) (*Leaf* 2)). In the end, we just need an instance for *Size Int* to compute the final result. Literals of type *Int* illustrate what we often call *opaque types*: those types that constitute the base of the universe and are *opaque* to the representation language.

2.2.2 EXPLICIT SUMS OF PRODUCTS

The other side of the coin is restricting the shape of the generic values to follow a *sums-of-products* format. This was first done by Löh and de Vries[23] in the `generics-sop` library. The main difference is in the introduction of *Codes*, that limit the structure of representations. If we had access to a representation of the *sum-of-products* structure of *Bin*, we could have defined our *gsize* function following an informal description: sum up the sizes of the fields inside a value, ignoring the constructor.

Unlike `GHC.Generics`, the representation of values is defined by induction on the *code* of a datatype, this *code* is a type-level list of lists of kind ***, whose semantics is consonant to a formula in disjunctive normal form. The outer list is interpreted as a sum and each of the inner lists as a product. This section provides an overview of `generics-sop` as required to understand the techniques we use in Chapter 3. We refer the reader to the original paper [23] for a more comprehensive explanation.

Using a *sum-of-products* approach one could write the same *gsize* function shown in Section 2.2.1 as easily as:

$$\begin{aligned} gsize &:: (\text{Generic}_{\text{sop}}\ a) \Rightarrow a \rightarrow \text{Int} \\ gsize &= \text{sum} \circ \text{elim}\ (\text{map}\ \text{size}) \circ \text{from}_{\text{sop}} \end{aligned}$$

Ignoring the details of *gsize* for a moment, let us focus just on its high level structure. Remembering that *from* now returns a *sum-of-products* view over the data, we are using an eliminator, *elim*, to apply a function to the fields of the constructor used to create a value of type *a*. This eliminator then applies *map size* to the fields of the constructor, returning something akin to a *[Int]*. We then *sum* them up to obtain the final size.

Codes consist of a type-level list of lists. The outer list represents the constructors of a type, and will be interpreted as a sum, whereas the inner lists are interpreted as the fields of the respective constructors, interpreted as products. The ' sign in the code

below marks the list as operating at the type-level, as opposed to term-level lists which exist at run-time. This is an example of Haskell’s *datatype* promotion [106].

```

678 type family   Codesop (a :: *) :: '[ * ]
680 type instance Codesop (Bin a) = '[ [a], [Bin a, Bin a]]

```

The *representation* is then defined by induction on *Code_{sop}* by the means of generalized *n*-ary sums, *NS*, and *n*-ary products, *NP*. With a slight abuse of notation, one can view *NS* and *NP* through the lens of the following type isomorphisms:

$$\begin{aligned}
 NS\ f\ [k_1, k_2, \dots] &\equiv f\ k_1\ :\!+\!:\ (f\ k_2\ :\!+\!:\ \dots) \\
 NP\ f\ [k_1, k_2, \dots] &\equiv f\ k_1\ :\!*\!:\ (f\ k_2\ :\!*\!:\ \dots)
 \end{aligned}$$

If we define *Rep_{sop}* to be *NS* (*NP* (*K1 R*)), where **data** *K1 R a* = *K1 a* is borrowed from *GHC.Generics*, we get exactly the representation that *GHC.Generics* issues for *Bin a*. Nevertheless, note how we already need the parameter *f* to pass *NP* to *NS* here.

$$\begin{aligned}
 Rep_{sop}\ (Bin\ a) &\equiv NS\ (NP\ (K1\ R))\ (Code_{sop}\ (Bin\ a)) \\
 &\equiv K1\ R\ a\ :\!+\!:\ (K1\ R\ (Bin\ a)\ :\!*\!:\ K1\ R\ (Bin\ a)) \\
 &\equiv Rep_{gen}\ (Bin\ a)
 \end{aligned}$$

It makes no sense to go through the trouble of adding the explicit *sums-of-products* structure to forget this information in the representation. Instead of piggybacking on *pattern functors*, we define *NS* and *NP* from scratch using *GADTs* [104]. By pattern matching on the values of *NS* and *NP* we inform the type checker of the structure of *Code_{sop}*.

```

688 data NS :: (k → *) → [k] → * where
689   Here  :: f k      → NS f (k ' :  ks)
690   There :: NS f ks → NS f (k ' :  ks)
694 data NP :: (k → *) → [k] → * where
695   ε      ::          NP f ' []
696   (×) :: f x → NP f xs → NP f (x ' :  xs)

```

Finally, since our atoms are of kind ***, we can use the identity functor, *I*, to interpret those and define the final representation of values of a type *a* under the *SOP* view:

```

697 type Repsop a = NS (NP I) (Codesop a)
newtype I (a :: *) = I { unI :: a }

```

To support the claim that one can define general combinators for working with these representations, let us look at *elim* and *map*, used to implement the *gsi* function in the

beginning of the section. The *elim* function just drops the constructor index and applies *f*, whereas the *map* applies *f* to all elements of a product.

```

elim :: (∀ k . f k → a) → NS f ks → a
elim f (Here x) = f x
elim f (There x) = elim f x
702 map :: (∀ k . f k → a) → NP f ks → [a]
map f ε      = []
map f (x × xs) = f x : map f xs

```

Reflecting on the current definition of *size* and comparing it to the `GHC.Generics` implementation of *size*, we see two improvements: (A) we need one fewer typeclass, *GSize*, and, (B) the definition is combinator-based. Considering that the generated *pattern functor* representation of a Haskell datatype will already be in a *sums-of-products*, we do not lose anything by enforcing this structure.

There are still downsides to this approach. A notable one is the need to carry constraints around: the actual *gsize* written with the `generics-sop` library and no sugar reads as follows.

```

711 gsize :: (Genericsop a , All2 Size (Codesop a)) ⇒ a → Int
      gsize = sum ∘ hcollapse
              ∘ hmap (Proxy :: Proxy Size) (mapIK size) ∘ fromsop

```

Where *hcollapse* and *hmap* are analogous to the *elim* and *map* combinators defined above. The *All2 Size (Code_{sop} a)* constraint tells the compiler that all of the types serving as atoms for *Code_{sop} a* are an instance of *Size*. Here, *All2 Size (Code_{sop} (Bin a))* expands to *(Size a , Size (Bin a))*. The *Size* constraint also has to be passed around with a *Proxy* for the eliminator of the *n*-ary sum. This is a direct consequence of a *shallow* encoding: since we only unfold one layer of recursion at a time, we have to carry proofs that the recursive arguments can also be translated to a generic representation. We can relieve this burden by recording, explicitly, which fields of a constructor are recursive or not, which is exactly how we start to shape `generics-mrsop` in Chapter 3.

2.2.3 DISCUSSION

Most other generic programming libraries follow a similar pattern of defining the *description* of a datatype in the provided uniform language by some type-level information, and two functions witnessing an isomorphism. The most important feature of such library is how this description is encoded and which are the primitive operations for constructing such encodings. Some libraries, mainly deriving from the SYB approach [49, 72], use the *Data* and *Typeable* typeclasses instead of static type-level information to provide generic functionality – these are a completely different strand of work from what we seek. The

	Pattern Functors	Codes
No Explicit Recursion	<code>GHC.Generics</code>	<code>generics-sop</code>
Simple Recursion	<code>regular</code>	
Mutual Recursion	<code>multirec</code>	

FIGURE 2.10: *Spectrum of static generic programming libraries*

main approaches that rely on type-level representations of datatypes are shown in Figure 2.10. These can be compared in their treatment of recursion and on their choice of type-level combinators used to represent generic values.

RECURSION STYLE. There are two ways to define the representation of values. Either we place explicit information about which fields of the constructors of the datatype in question are recursive or we do not.

If we do not mark recursion explicitly, *shallow* encodings are the easier option, where only one layer of the value is turned into a generic form by a call to *from*. This is the kind of representation we get from `GHC.Generics`. The other side of the spectrum would be the *deep* representation, in which the entire value is turned into the representation that the generic library provides in one go.

Marking the recursion explicitly, like in `regular` [76], allows one to choose between *shallow* and *deep* encodings at will. These representations are usually more involved as they need an extra mechanism to represent recursion. In the *Bin* example, the description of the *Bin* constructor changes from “this constructor has two fields of the *Bin a* type” to “this constructor has two fields in which you recurse”. Therefore, a *deep* encoding requires some explicit *least fixpoint* combinator – usually called *Fix* in Haskell.

Depending on the use case, a shallow representation might be more efficient if only part of the value needs to be inspected. On the other hand, deep representations are sometimes easier to use, since the conversion is performed in one go, and afterwards one only has to work with the constructs from the generic library.

The fact that we mark explicitly when recursion takes place in a datatype gives some additional insight into the description. Some functions really need the information about which fields of a constructor are recursive and which are not, like the generic *map* and the generic *Zipper*. This additional power has also been used to define regular expressions over Haskell datatypes [92], for example.

PATTERN FUNCTORS VERSUS CODES. Most generic programming libraries build their type-level descriptions out of three basic combinators: (1) *constants*, which indicate a

757 type is atomic and should not be expanded further; (2) *products* (usually written as
 758 `:*`) which are used to build tuples; and (3) *sums* (usually written as `:+`) which en-
 759 code the choice between constructors. The *Rep* (*Bin a*) shown before is expressed in
 760 this form. Note, however, that there is no restriction on *how* these can be combined.
 761 These combinators are usually referred to as *pattern functors*. The *pattern functor*-based
 762 libraries are too permissive though, for instance, *K1 R Int* `:*` *Maybe* is a perfectly valid
 763 `GHC.Generics pattern functor` but will break generic functions, i.e., *Maybe* is not a
 764 combinator.

765 In practice, one can always use a sum of products to represent a datatype – a sum
 766 to express the choice of constructor, and within each constructor a product to declare
 767 which fields you have. The `generic-sop` library [23] explicitly uses a list of lists of
 768 types, the outer one representing the sum and each inner one thought of as products.

769 $Code_{\text{sop}} (Bin\ a) = '[a], '[Bin\ a, Bin\ a]$

770 The shape of this description follows more closely the shape of Haskell datatypes,
 771 and make it easier to implement generic functionality.

772 Note how the *codes* are different than the *representation*. The latter being defined
 773 by induction on the former. This is quite a subtle point and it is common to see both
 774 terms being used interchangeably. Here, the *representation* is mapping the *codes*, of kind
 775 `'[*]`, into `*`. The *code* can be seen as the format that the *representation* must adhere
 776 to. Previously, in the pattern functor approach, the *representation* was not guaranteed
 777 to have a certain structure. The expressivity of the language of *codes* is proportional to
 778 the expressivity of the combinators the library can provide.

779



780

781 GENERIC PROGRAMMING WITH 782 MUTUALLY RECURSIVE TYPES

783 The syntax of many programming languages is expressed through a mutually recursive
784 family of datatypes. Before writing a generic differencing algorithm we need to be able
785 to program generically over mutually recursive families of datatypes. Consider Haskell
786 itself, a **do** block constructs an expression, even though the **do** block itself is composed
787 by a list of statements which may include expressions.

```
788 data Expr = ... | Do [Stmt] | ...  
       data Stmt = Assign Var Expr | Let Var Expr
```

789 Another example is found in HTML and XML documents. Which are easily de-
790 scribed by a Rose tree, which albeit being a nested type, is naturally encoded in the
791 mutually recursive family of datatypes below.

```
792 data Rose a = Fork a [Rose a]  
       data [] a = [] | a : [a]
```

793 The mutual recursion becomes apparent once one instantiates *a* to some ground
794 type, for instance:

```
795 data RoseI = Fork Int ListI  
       data ListI = Nil | RoseI : ListI
```

Working with generic mutually recursive families in Haskell, however, is a non-trivial task. The best solution at the time of writing is the `multirec` [105] library, which is unfortunately unfit for writing complex programs – the lack of a combinator-based approach to generic programming and the pattern functor (Section 2.2.1) approach makes it hard to write involved algorithms.

This meant we had to engineer new generic programming libraries to tackle the added complexity of mutual recursion. We have devised two different ways of doing so. First, we wrote the `generics-mrsop` [70] library, which combines a combinator based (Section 2.2.2) approach to generic programming with mutually recursive types. In fact, `generics-mrsop` lies in the intersection of `multirec` and the more modern `generics-sop` [23]. It is worth noting that neither of the aforementioned libraries *compete* with our work. We extend both in orthogonal directions, resulting in a new design altogether, that takes advantage of some modern Haskell extensions which the authors of the previous work could not employ.

The `generics-mrsop` library, Section 3.1, was a conceptual success. It enabled us to prototype and tweak the algorithms discussed in Chapter 4 and Chapter 5 with ease. Yet, a memory leak in the Glasgow Haskell Compiler¹ made it unusable for encoding real programming languages such as those in the `language-python` or `language-java` packages. This frustrating outcome meant that a different approach – which did not rely as heavily on type families – was necessary to look at real-world software version control conflict data.

Turns out we can sacrifice the sums-of-products structure of `generics-mrsop`, significantly decreasing the reliance of type families, but still maintaining a combinator-based approach, which still enables us to write the algorithms underlying the `hdiff` tool (Chapter 5). This lead us to develop the `generics-simplistic` library, Section 3.2, which still maintains a list of the types that belong in the family, but does not record their internal sum-of-products structure.

This chapter, then, is concerned with explaining our work extending the existing generic programming capabilities of Haskell to support mutually recursive types. We introduce two conceptually different approaches, but with similar expressivity. In Section 3.1 we explore the `generics-mrsop` library. With its ability of representing explicit sums of products we are able to illustrate the `gdif` [51] differencing algorithm, which follows the classical tree-edit distance but in a typed fashion. Then, in Section 3.2, we explore the `generics-simplistic` library, which works on the pattern functor spectrum of generic programming.

¹ <https://gitlab.haskell.org/ghc/ghc/issues/17223> and <https://gitlab.haskell.org/ghc/ghc/issues/14987>

831 3.1 THE GENERICS-MRSOP LIBRARY

832 The `generics-mrsop` library is an intersection of the `multirec` and `generics-sop`
 833 libraries. It uses explicit codes in the sums of products style to guide the representation of
 834 datatypes. This enables a simple explicit fixpoint construction and a variety of recursion
 835 schemes, which makes the development of generic programs fairly straightforward.

836 3.1.1 EXPLICIT FIXPOINTS WITH CODES

837 Introducing information about the recursive positions in a type requires more expressive
 838 codes than in Section 2.2.2. Where our *codes* were a list of lists of types, which could
 839 be anything, we now have a list of lists of *Atom*, which maintains information about
 840 whether a position is recursive or not.

```
841 data Atom = I | KInt | ...
type family Codefix (a :: *) :: '[Atom]
type instance Codefix (Bin Int) = '[KInt], '[I, I]
```

842 Here, *I* is used to mark the recursive positions and *KInt*, ... are codes for a prede-
 843 termined selection of primitive types, which we refer to as *opaque types*. Favoring the
 844 simplicity of the presentation, we will stick with only hard coded *Int* as the only opaque
 845 type in the universe. Later on, in Section 3.1.2.1, we parameterize the whole develop-
 846 ment by the choice of opaque types.

847 We can no longer represent polymorphic types in this universe – the *codes* them-
 848 selves are not polymorphic. Back in Section 2.2.2 we have defined *Code_{sop}* (*Bin a*), and
 849 this would work for any *a*. The lack of polymorphism might seem like a disadvantage
 850 at first, but if we are interested in deep generic representations, it is actually an advan-
 851 tage, as it allows us to have a deep conversion for free as we do not need to carry *Generic*
 852 constraints around. That is, say we want to deeply convert a value of type *Bin a* to its
 853 generic representation polymorphically on *a*. We can only do so if we have access to the
 854 *Code_{sop}* *a*, which comes from knowing *Generic a*. By specifying the types involved be-
 855 forehand, we are able to get by without having to carry all of the constraints we needed
 856 in, for instance, *gsiz*e at the end of Section 2.2.2. The main benefit is in the simplicity of
 857 combinators we will define in Section 3.1.2.2.

858 The Rep_{fix} datatype is similar to the Rep_{sop} , but uses an additional layer that maps
 859 an Atom into $*$, denoted NA . Since an atom can be either an opaque type, known statically,
 860 or some type that must be placed in a recursive position later on, we need just one
 861 parameter in NA .

```

862 data NA :: * → Atom → * where
      NA_I :: x → NA x I
      NA_K :: Int → NA x KInt
  newtype Rep_fix a x = Rep { unRep :: NS (NP (NA x)) (Code_fix a) }

```

863 The $\text{Generic}_{\text{fix}}$ typeclass, below, witnesses the isomorphism between ordinary types
 864 and their deep sums-of-products representation. Similarly to the other generic type-
 865 classes out there, it contains just the familiar to_{fix} and from_{fix} components. We illustrate
 866 part of the instance that witnesses that Bin Int has a generic representation below. We
 867 omit the to_{fix} function as it is the opposite of from_{fix} .

```

      class Generic_fix a where
        from_fix :: a → Rep_fix a a
        to_fix   :: Rep_fix a a → a
  868 instance Generic_fix (Bin Int) where
        from_fix (Leaf x) = Rep ( Here (NA_K x × ε) )
        from_fix (Bin l r) = Rep ( There (Here (NA_I l × NA_I r × ε)) )

```

869 It is an interesting exercise to implement the Functor instance for $(\text{Rep}_{\text{fix}} a)$ – where
 870 it can be seen that we were only able to lift it to a functor by recording the information
 871 about the recursive positions. Otherwise, there would be no easy way of knowing where
 872 to apply f when defining $\text{fmap } f$.

873 Nevertheless, working directly with Rep_{fix} is hard – we need to pattern match on
 874 There and Here , whereas we actually want to have the notion of *constructor* for the
 875 generic setting too! The main advantage of the *sum-of-products* structure is to allow
 876 a user to pattern match on generic representations just like they would on values of the
 877 original type, contrasting with `GHC.Generics`. One can precisely state that a value of
 878 a representation is composed by a choice of constructor and its respective product of
 879 fields by the View type. This *view* pattern [101, 61] is common in dependently typed
 880 programming.

```

      data Nat = Z | S Nat
  881 data View :: [[Atom]] → * → * where
        Tag :: Constr n t → NP (NA x) (Lkup t n) → View t x

```

882 A value of $\text{Constr } n \text{ sum}$ is a proof that n is a valid constructor for sum , stating that
 883 $n < \text{length sum}$. Lkup performs list lookup at the type-level. To improve type error mes-

sages, we generate a *TypeError* whenever we reach a given index n that is out of bounds. Interestingly, our design guarantees that this case is never reached by *Constr*.

```

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

type family Lkup (ls :: [k]) (n :: Nat) :: k where
  Lkup '[] _ = TypeError "Index out of bounds"
  Lkup (x : xs) 'Z = x
  Lkup (x : xs) ('S n) = Lkup xs n

```

With the help of *sop* and *inj*, declared below, we are able to pattern match and inject into generic values. Unfortunately, matching on *Tag* directly can be cumbersome, but we can always use pattern synonyms [84] to circumvent that. For example, the synonyms below describe the constructors *Bin* and *Leaf*.

```

pattern (Pat Leaf) x = Tag CZ (NAK x × ε)
pattern (Pat Bin) l r = Tag (CS CZ) (NAI l × NAI r × ε)

inj :: View sop x → Repfix sop x
sop :: Repfix sop x → View sop x

```

Having the core of the *sums-of-products* universe defined, we can turn our attention to writing the combinators that the programmer will use. These will naturally be defined by induction on the *Code_{fix}* instead of having to rely on instances, like in Section 2.2.1. For instance, lets look at *compos*, which applies a function f everywhere on the recursive structure.

```

compos :: (Genericfix a) ⇒ (a → a) → a → a
compos f = tofix ∘ fmap f ∘ fromfix

```

Although more interesting in the mutually recursive setting, Section 3.1.2, we can illustrate its use for traversing a tree and adding one to its leaves. This example is a bit convoluted, since one could get the same result by simply writing *fmap (+1)* :: *Bin Int* → *Bin Int*, but shows the intended usage of the *compos* combinator just defined.

```

example :: Bin Int → Bin Int
example (Leaf n) = Leaf (n + 1)
example x       = compos example x

```

It is worth noting the *catch-all* case, allowing one to focus only on the interesting patterns and using a default implementation everywhere else, which is convenient when the datatypes in question are large and might change often.

```

crush :: (Genericfix a) ⇒ (∀ x . Int → b) → ([b] → b) → a → b
crush k cat = crushFix ∘ deepFrom
  where crushFix :: Fix (Repfix a) → b
        crushFix = cat ∘ elimNS (elimNP go) ∘ unFix
        go (NAI x) = crushFix x
        go (NAK i) = k i

```

FIGURE 3.1: *Generic crush combinator*

906 CONVERTING TO A DEEP REPRESENTATION. The *from_{fix}* function still returns a shallow representation. But by constructing the least fixpoint of *Rep_{fix}* *a* we can easily obtain
 907 the deep encoding for free, by recursively translating each layer of the shallow encoding.
 908

```

newtype Fix f = Fix {unFix :: f (Fix f)}
909 deepFrom :: (Genericfix a) ⇒ a → Fix (Repfix a)
    deepFrom = Fix ∘ fmap deepFrom ∘ fromfix

```

910 So far, we handle the same class of types as the *regular* [76] library, but we require
 911 the representation to follow a sums of products structure by the means of *Code_{fix}*. Those
 912 types are guaranteed to have an initial algebra, and indeed, the generic catamorphism is
 913 defined as expected:

```

914 fold :: (Repfix a b → b) → Fix (Repfix a) → b
    fold f = f ∘ fmap (fold f) ∘ unFix

```

915 Some functions may consume a value and produce a single value, but do not need
 916 the full expressivity of *fold*. Instead, if we know how to consume the opaque types and
 917 combine those results, we can consume any *Generic_{fix}* type using *crush*, which is defined
 918 in Figure 3.1. The behavior of *crush* is defined by (1) how to turn atoms into the output
 919 type *b* – in this case we only have integer atoms, and thus we require an *Int* → *b* function
 920 – and (2) how to combine the values bubbling up from each member of a product.

921 Finally, we come full circle to our running *gsize* example as it was promised in the
 922 introduction. This is noticeably the smallest implementation so far, and very straight to
 923 the point.

```

924 gsize :: (Genericfix a) ⇒ a → Int
    gsize = crush (const 1) sum

```

925 At this point we have combined the insight from the *regular* library of keeping
 926 track of recursive positions with the convenience of the *generics-sop* for enforcing

a specific *normal form* on representations. By doing so, we were able to provide a deep encoding for free. This essentially frees us from the burden of maintaining complicated constraints needed for handling the types within the topmost constructor. The information about the recursive position allows us to write neat combinators like *crush* and *compos* together with a convenient *View* type for easy generic pattern matching. The only thing keeping us from handling real life applications is the limited form of recursion.

3.1.2 MUTUAL RECURSION

Conceptually, going from regular types (Section 3.1.1) to mutually recursive families is simple. We just need to reference not only one type variable, but one for each element in the family. This is usually [54, 4] done by adding an index to the recursive positions to represents each member of the family. As a running example, we use the familiar *rose tree* family.

```
data Rose a = Fork a [Rose a]
data [] a = [] | a : [a]
```

The previously introduced *Code_{fix}*, Section 3.1.1, is not expressive enough to describe this datatype. In particular, when we try to write *Code_{fix} (Rose Int)*, there is no immediately recursive appearance of *Rose* itself, so we cannot use the atom *I* in that position. Furthermore *[Rose a]* is not an opaque type either, so we cannot use any of the other combinators provided by *Atom*. We would like to record information about *Rose Int* referring to itself via another datatype.

Our solution is to move from codes of datatypes to *codes for families of datatypes*. We no longer talk about *Code_{fix} (Rose Int)* or *Code_{fix} [Rose Int]* in isolation. Codes only make sense within a family, that is, a list of types. Hence, we talk about the codes of the two types in the family: *Code_{mrec} '[Rose Int, [Rose Int]]*. Then we extend the language of *Atoms* by appending to *I* a natural number which specifies the member of the family to recurse into:

```
data Atom = I Nat | KInt | ...
```

The code of this recursive family of datatypes can be described as:

```
type FamRose = '[Rose Int, [Rose Int]]
type Codemrec FamRose = '[ '[KInt, I (S Z)]
                        , '[[], '[I Z, I (S Z)]]
                        ]
```

Let us have a closer look at the code for *Rose Int*, which appears in the first place in the list. There is only one constructor which has an *Int* field, represented by *KInt*, and another in which we recurse via the second member of our family (since lists are 0-indexed, we represent this by *S Z*). Similarly, the second constructor of *[Rose Int]* points back to both *Rose Int* using *I Z* and to *[Rose Int]* itself via *I (S Z)*.

Having settled on the definition of *Atom*, we now need to adapt *NA* to the new *Atoms*. To interpret any *Atom* into $*$, we need a way assign values to the different recursive positions. This information is given by an additional type parameter φ that maps natural numbers into types.

```
data NA :: (Nat → *) → Atom → * where
  NA_I :: φ n → NA φ (I n)
  NA_K :: Int → NA φ KInt
```

This additional φ naturally bubbles up to *Rep_{mrec}*.

```
type Repmrec (φ :: Nat → *) (c :: [[Atom]]) = NS (NP (NA φ)) c
```

The only piece missing here is tying the recursive knot. If we want our representation to describe a family of datatypes, the obvious choice for φn is to look up the type at index n in *FamRose*. In fact, we are simply performing a type-level lookup in the family, so we can reuse the *Lkup* from Section 3.1.1.

In principle, this is enough to provide a ground representation for the family of types. Let *fam* be a family of types, like *[Rose Int , [Rose Int]]*, and *codes* the corresponding list of codes. Then the representation of the type at index *ix* in the list *fam* is given by:

```
Repmrec (Lkup fam) (Lkup codes ix)
```

This definition states that to obtain the representation of the type at index *ix*, we first lookup its code. Then, in the recursive positions we interpret each *I n* by looking up the type at that index in the original family. This gives us a *shallow* representation.

Unfortunately, Haskell only allows saturated, that is, fully-applied type families. Hence, we cannot partially apply *Lkup* like we did it in the example above. As a result, we need to introduce an intermediate datatype *El*,

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

The representation of the family *fam* at index *ix* is thus given in terms of *El*, which can be partially applied, *Rep_{mrec} (El fam) (Lkup codes ix)*. We only need to use *El* in

the first argument, because that is the position in which we require partial application. The second position has *Lkup* already fully-applied, and can stay as is.

We still have to relate a family of types to their respective codes. As in other generic programming approaches, we want to make their relation explicit. The *Family* typeclass below realizes this relation, and introduces functions to perform the conversion between our representation and the actual types. Using *El* here spares us from using a proxy for *fam* in *from_{mrec}* and *to_{mrec}*:

```
class Family (fam :: [ * ]) (codes :: [[Atom]]) where
  frommrec :: SNat ix → El fam ix → Repmrec (El fam) (Lkup codes ix)
  tomrec   :: SNat ix → Repmrec (El fam) (Lkup codes ix) → El fam ix
```

One of the differences between other approaches and ours is that we do not use an associated type to define the *codes* for the family *fam*. One of the reasons to choose this path is that it alleviates the burden of writing the longer *Code_{mrec} fam* every time we want to refer to *codes*. Furthermore, there are types like lists which appear in many different families, and in that case it makes sense to speak about a relation instead of a function.

Since now *from_{mrec}* and *to_{mrec}* operate on families, we have to specify how to translate *each* of the members of the family *to* and *from* their generic representation. This translation needs to know which is the index of the datatype we are converting between in each case, hence the additional singleton *SNat ix* parameter. Pattern matching on this singleton [26] type informs the compiler about the shape of the *Nat* index. Its definition is:

```
data SNat (n :: Nat) where
  SZ :: SNat 'Z
  SS :: SNat n → SNat ('S n)
```

Which in turn, enables us to write the definition of *from_{mrec}* for the family of rose trees.

```
-- First type in the family
frommrec SZ (El (Fork x ch)) = Rep (Here (NAK x × NAI ch × ε))
-- Second type in the family
frommrec (SS SZ) (El [ ])      = Rep (Here ε)
frommrec (SS SZ) (El (x : xs)) = Rep (There (Here (NAI x × NAI xs × ε)))
```

By pattern matching on the index, the compiler knows which family member to expect as a second argument. This then allows the pattern matching on the *El* to typecheck.

The limitations of the Haskell type system lead us to introduce *El* as an intermediate datatype. Our *from_{mrec}* function does not take a member of the family directly, but

an *El*-wrapped one. However, to construct that value, *El* needs to know its parameters, which amounts to knowing the family we are embedding our type into and the index in that family. Those values are not immediately obvious, but we can use Haskell's visible type application [27] to work around it. The *into* function injects a value into the corresponding *El*:

```

1013   into :: ∀ fam ty ix . (ix ~ Idx ty fam , Lkup fam ix ~ ty) ⇒ ty → El fam ix
1014   into = El
1018   intoRose :: Rose Int → El RoseFam 'Z
1019   intoRose = into @FamRose

```

Idx, here, is a closed type family implementing the inverse of *Lkup*, that is, obtaining the index of the type *ty* in the list *fam*. Using this function we can turn a *[Rose Int]* into its generic representation by writing *from_{mrec} ∘ into @FamRose*. The type application *@FamRose* is responsible for fixing the mutually recursive family we are working with, which allows the type checker to reduce all the constraints and happily inject the element into *El*.

DEEP REPRESENTATION. In Section 3.1.1 we have described a technique to derive deep representations from shallow representations. We can play a very similar trick here. The main difference is the definition of the least fixpoint combinator, which receives an extra parameter of kind *Nat* indicating which *code* to use first:

```

1029   newtype Fix (codes :: [[[Atom]]]) (ix :: Nat)
1030   = Fix {unFix :: Repmrec (Fix codes) (Lkup codes ix)}

```

Intuitively, since now we can recurse on different positions, we need to keep track of the representations for all those positions in the type. This is the job of the *codes* argument. Furthermore, our *Fix* does not represent a single datatype, but rather the *whole* family. Thus, we need each value to have an additional index to declare on which element of the family it operates.

As in the previous section, we can obtain the deep representation by iteratively applying the shallow representation. Earlier we used *fmap* since the *Rep_{fix}* type was a functor. *Rep_{mrec}* on the other hand cannot be given a *Functor* instance, but we can still define a similar function *mapRep*,

```

1039   mapRep :: (∀ ix . ϕ1 ix → ϕ2 ix) → Repmrec ϕ1 c → Repmrec ϕ2 c

```

This signature tells us that if we want to change the *ϕ₁* argument in the representation, we need to provide a natural transformation from *ϕ₁* to *ϕ₂*, that is, a function

1042 which works over each possible index this φ_1 can take and does not change this index.
 1043 This follows from φ_1 having kind $\text{Nat} \rightarrow *$.

1044
$$\begin{aligned} \text{deepFrom} &:: \text{Family fam codes} \Rightarrow \text{El fam ix} \rightarrow \text{Fix} (\text{Rep}_{\text{mrec}} \text{ codes ix}) \\ \text{deepFrom} &= \text{Fix} \circ \text{mapRec} \text{ deepFrom} \circ \text{from}_{\text{mrec}} \end{aligned}$$

1045 ONLY WELL-FORMED REPRESENTATIONS ARE ACCEPTED. At first glance, it may seem
 1046 like the *Atom* datatype gives too much freedom: its *I* constructor receives a natural num-
 1047 ber, but there is no apparent static check that this number refers to an actual mem-
 1048 ber of the recursive family we are describing. For example, the list of codes given by
 1049 `'[['[KInt, I (S (S Z))]]]` is accepted by the compiler although it does not represent any
 1050 family of datatypes.

1051 A direct solution to this problem is to introduce yet another index, this time in the
 1052 *Atom* datatype, which specifies which indices are allowed. The *I* constructor is then
 1053 refined to take not any natural number, but only those which lie in the range – this is
 1054 usually known as *Fin n*.

1055
$$\text{data Atom } (n :: \text{Nat}) = \text{I} (\text{Fin } n) \mid \text{KInt} \mid \dots$$

1056 The lack of dependent types makes this approach very hard, in Haskell. We would
 1057 need to carry around the inhabitants *Fin n* and define functionality to manipulate them,
 1058 which would greatly hinder the usability of the library.

1059 By looking a bit more closely, we find that we are not losing any type-safety by al-
 1060 lowing codes which reference an arbitrary number of recursive positions. Users of our
 1061 library are allowed to write the previous ill-defined code, but when trying to write *val-*
 1062 *ues* of the representation of that code, the *Lkup* function detects the out-of-bounds index,
 1063 raising a type error and preventing the program from compiling in the first place, instead
 1064 of crashing at run-time.

1065 3.1.2.1 PARAMETERIZED OPAQUE TYPES

1066 Up to this point we have considered *Atom* to include a predetermined selection of *opaque*
 1067 *types*, such as *Int*, each of them represented by one of the constructors other than *I*. This
 1068 is far from ideal, for two conflicting reasons:

- 1069 a) The choice of opaque types might be too narrow. For example, the user of our
- 1070 library may decide to use *ByteString* in their datatypes. Since that type is not cov-
 1071 ered by *Atom*, nor by our generic approach, this implies that `generics-mrsop`
 1072 becomes useless to them.

1073 b) The choice of opaque types might be too wide. If we try to encompass any possible
 1074 situation, we end up with a huge *Atom* type. But for a specific use case, we might
 1075 be interested only in *Ints* and *Floats*, so why bother ourselves with possibly ill-
 1076 formed representations and pattern matches which should never be reached?

1077 Our solution is to *parameterize* *Atom*, giving users the choice of opaque types:

1078 **data** *Atom* *kon* = *I* *Nat* | *K* *kon*

1079 For example, if we only want to deal with numeric opaque types, we can write:

1080 **data** *NumericK* = *KInt* | *KInteger* | *KFloat*
 1081 **type** *NumericAtom* = *Atom* *NumericK*

1081 The representation of codes must be updated to reflect the possibility of choosing
 1082 different sets of opaque types. The *NA* datatype in this final implementation provides
 1083 two constructors, one per constructor in *Atom*. The *NS* and *NP* datatypes do not require
 1084 any change.

1085 **data** *NA* :: (*kon* → *) → (*Nat* → *) → *Atom* *kon* → * **where**
 NA_I :: φ *n* → *NA* κ φ (*I* *n*)
 NA_K :: κ *k* → *NA* κ φ (*K* *k*)
type *Rep_{mrec}* (κ :: *kon* → *) (φ :: *Nat* → *) (*c* :: [[*Atom* *kon*]]) = *NS* (*NP* (*NA* κ φ)) *c*

1086 The *NA_K* constructor in *NA* makes use of an additional argument κ . The problem
 1087 is that we are defining the code for the set of opaque types by a specific kind, such as
 1088 *Numeric* above. On the other hand, values which appear in a field must have a type
 1089 whose kind is *. Thus, we require a mapping from each of the codes to the actual opaque
 1090 type they represent, this is exactly the *opaque type interpretation* κ . Here is the datatype
 1091 interpreting *NumericK* into ground types:

1092 **data** *NumericI* :: *NumericK* → * **where**
 IInt :: *Int* → *NumericI* *KInt*
 IFloat :: *Float* → *NumericI* *KFloat*

1093 The last piece of our framework which has to be updated to support different sets
 1094 of opaque types is the *Family* typeclass, as given in Figure 3.2. This typeclass provides
 1095 an interesting use case for the new dependent features in Haskell; both κ and *codes* are
 1096 parameterized by an implicit argument *kon* which represents the set of opaque types.

1097 We stress that the parametrization over opaque types does *not* mean that we can use
 1098 only closed universes of opaque types. It is possible to provide an *open* representation
 1099 by choosing (*) – the whole kind of Haskell’s ground types – as argument to *Atom*. As

```

class Family ( $\kappa :: \text{kon} \rightarrow *$ ) ( $\text{fam} :: [ * ]$ ) ( $\text{codes} :: [[\text{Atom kon}]]$ ) where
  frommrec ::  $\text{SNat } ix \rightarrow \text{El } \text{fam } ix \rightarrow \text{Rep}_{\text{mrec}} \kappa (\text{El } \text{fam}) (\text{Lkup codes } ix)$ 
  tomrec   ::  $\text{SNat } ix \rightarrow \text{Rep}_{\text{mrec}} \kappa (\text{El } \text{fam}) (\text{Lkup codes } ix) \rightarrow \text{El } \text{fam } ix$ 

```

FIGURE 3.2: *Family* typeclass with support for different opaque types

1100 a consequence, the interpretation ought to be of kind $* \rightarrow *$, as given by *Value*, below.
 1101 To use $(*)$ as an argument to a type, we must enable the `TypeInType` language extension [102, 103].
 1102

```

data Value ::  $* \rightarrow *$  where
  Value ::  $t \rightarrow \text{Value } t$ 

```

1104 3.1.2.2 SELECTION OF USEFUL COMBINATORS

1105 The advantages of a *code based* approach to generic programming becomes evident when
 1106 we look at the generic combinators that `generics-mrsop` provides. We refer the reader
 1107 to the actual documentation for a comprehensive list. Here we look at a selection of use-
 1108 ful functions in their full form. Let us start with the bifunctionality of *Rep_{mrec}*:

```

bimapRep :: ( $\forall k . \kappa_1 k \rightarrow \kappa_2 k$ )  $\rightarrow$  ( $\forall ix . \varphi_1 ix \rightarrow \varphi_2 ix$ )
   $\rightarrow \text{Rep}_{\text{mrec}} \kappa_1 \varphi_1 c \rightarrow \text{Rep}_{\text{mrec}} \kappa_2 \varphi_2 c$ 
bimapRep  $f_k f_l = \text{mapNS } (\text{mapNP } (\text{mapNA } f_l f_l))$ 

```

1110 To destruct a *Rep_{mrec}* $\kappa \varphi c$ we need a way for eliminating every recursive position
 1111 or opaque type inside the representation and a way of combining these results.

```

elimRep :: ( $\forall k . \kappa k \rightarrow a$ )  $\rightarrow$  ( $\forall ix . \varphi ix \rightarrow a$ )  $\rightarrow$  ( $[a] \rightarrow b$ )  $\rightarrow \text{Rep}_{\text{mrec}} \kappa \varphi c \rightarrow b$ 
elimRep  $f_k f_l \text{ cat} = \text{elimNS cat } (\text{elimNP } (\text{elimNA } f_k f_l))$ 

```

1113 Another useful operator, particularly when combined with *bimapRep* is the *zipRep*,
 1114 that works just like a regular *zip*. Our *zipRep* attempts to put two values of a representa-
 1115 tion “side-by-side”, as long as they are constructed with the same injection into the n -ary
 1116 sum, *NS*.

```

zipRep ::  $\text{Rep}_{\text{mrec}} \kappa_1 \varphi_1 c \rightarrow \text{Rep}_{\text{mrec}} \kappa_2 \varphi_2 c \rightarrow \text{Maybe } (\text{Rep}_{\text{mrec}} (\kappa_1 :*: \kappa_2) (\varphi_1 :*: \varphi_2) c)$ 
zipRep  $r s = \text{case } (\text{sop } r, \text{sop } s) \text{ of}$ 
  ( $\text{Tag } cr pr, \text{Tag } cs ps$ )  $\rightarrow \text{case testEquality } cr \text{ of}$ 
    Just Refl  $\rightarrow \text{inj } cr <\$> \text{zipWithNP zipAtom } pr ps$ 

```

```

geq :: (EqHO  $\kappa$ , Family  $\kappa$  fam codes)  $\Rightarrow$  ( $\forall k$  .  $\kappa k \rightarrow \kappa k \rightarrow \text{Bool}$ )
     $\rightarrow$  El fam ix  $\rightarrow$  El fam ix  $\rightarrow \text{Bool}$ 
geq eqK x y = go (deepFrom x) (deepFrom y)
  where go (Fix x) (Fix y)
        = maybe False (elimRep (uncurry eqK) (uncurry go) and) $ zipRep x y

```

FIGURE 3.3: Generic equality

1118 We use *testEquality* from *Data.Type.Equality* to check for type index equality and
 1119 inform the compiler of that fact by matching on *Refl*.

1120 Finally, we can start assembling these building blocks into more practical functional-
 1121 ity. Figure 3.3 shows the definition of generic equality using *generics-mrsop*, where
 1122 the *EqHO* typeclass is a lifted version of *Eq*, for types of kind $k \rightarrow *$, defined below. The
 1123 library also provide *ShowHO*, the *Show* counterpart.

```

1124 class EqHO (f :: a  $\rightarrow$  *) where
    eqHO ::  $\forall x$  . f x  $\rightarrow$  f x  $\rightarrow \text{Bool}$ 

```

1125 We decided to provide a custom equality in *generics-mrsop* for two main reasons.
 1126 Firstly, when we started developing the library the *-XQuantifiedConstraints* [16]
 1127 extension was not completed. Yet, once quantified constraints were available in Haskell
 1128 we wrote *generics-mrsop-2.2.0* using the extension and defining *EqHO f* as a syn-
 1129 onym to $\forall x \circ \text{Eq} (f x)$. Developing applications on top of *generics-mrsop* became
 1130 more difficult. The user now would have to reason about and pass around complicated
 1131 constraints down datatypes and auxiliary functions. Moreover, our use case was very
 1132 simple, not extracting any of the advantages of quantified constraints. Eventually we
 1133 decided to rollback to the lifted *EqHO* presented above in *generics-mrsop-2.3.0*.

1134 As presented so far, we have all the necessary tools to encode our first differencing
 1135 attempt, shown in Chapter 4 of this thesis. The next sections discusses some aspects that,
 1136 albeit not directly required for understanding the remainder of this thesis, are interesting
 1137 in their own right and round off the presentation of *generics-mrsop* as a library.

1138 3.1.3 PRACTICAL FEATURES

1139 The development of the *generics-mrsop* library started primarily to enable us to write
 1140 *hdiff* Chapter 5 possible. This was a great expressivity test for our generic program-
 1141 ming library and led us to develop overall useful features that, although not novel, make
 1142 the adoption of a generic programming library much more likely. This section is a small

1143 tutorial into two important practical features of `generics-mrsop` and documents the
1144 engineering effort that was put in the library.

1145 3.1.3.1 TEMPLATE HASKELL

1146 Having a convenient and robust way to get the *Family* instance for a given selection of
1147 datatypes is paramount for the usability of our library. In a real scenario, a mutually
1148 recursive family may consist of many datatypes with dozens of constructors. Sometimes
1149 these datatypes are written with parameters, or come from external libraries.

1150 Our goal here is to automate the generation of *Family* instances under all those cir-
1151 cumstances using *Template Haskell* [95]. From the programmers' point of view, they
1152 only need to call *deriveFamily* with the topmost (that is, the first) type of the family. For
1153 example:

```
1154 data Exp var = ...
data Stmt var = ...
data Prog var = ...
deriveFamily [t|Prog String|]
```

1155 The *deriveFamily* takes care of unfolding the (type-level) recursion until it reaches
1156 a fixpoint. In this case, the type synonym *FamProgString* = '[*Prog String* , ...]' will
1157 be generated, together with its *Family* instance. Optionally, one can also pass along a
1158 custom function to decide whether a type should be considered opaque. By default, it
1159 uses a selection of Haskell built-in types as opaque types.

1160 UNFOLDING THE FAMILY The process of deriving a whole mutually recursive family
1161 from a single member is conceptually divided into two disjoint processes. First we repeat-
1162 edly unfold all definitions and follow all the recursive paths until we reach a fixpoint. At
1163 that moment we know that we have discovered all the types in the family. Second, we
1164 translate the definition of those types to the format our library expects. During the un-
1165 folding process we keep a key-value map in a *State* monad, keeping track of three things:
1166 the types we have seen; the types we have seen *and* processed; and the indices of those
1167 within the family.

1168 Let us illustrate this process in a bit more detail using our running example of a
1169 mutually recursive family and consider what happens within *Template Haskell* when it
1170 starts unfolding the *deriveFamily* clause.

```
1171 data Rose a = Fork a [Rose a]
data [a] = [] | a : [a]
deriveFamily [t|Rose Int|]
```

1172 The first thing that happens is registering that we seen the type *Rose Int*. Since it is
 1173 the first type to be discovered, it is assigned index zero within the family. Next we need
 1174 to reify the definition of *Rose*. At this point, we query *Template Haskell* for the definition,
 1175 and we obtain `data Rose x = Fork x [Rose x]`. Since *Rose* has kind $* \rightarrow *$, it cannot be
 1176 directly translated – our library only supports ground types, which are those with kind
 1177 $*$. But we do not need a generic definition for *Rose*, we just need the specific case where
 1178 $x = \text{Int}$. Essentially, we just apply the reified definition of *Rose* to *Int* and β -reduce it,
 1179 giving us `Fork Int [Rose Int]`.

1180 The next processing step is looking into the types of the fields of the (single) con-
 1181 structor *Fork*. First we see *Int* and decide it is an opaque type, say *KInt*. Second, we see
 1182 `[Rose Int]` and notice it is the first time we see this type. Hence, we register it with a
 1183 fresh index, *S Z* in this case. The final result for *Rose Int* is `'['[K KInt, I (S Z)]]`.

1184 We now go into `[Rose Int]` for processing. Once again we need to perform some
 1185 amount of β -reduction at the type-level before inspecting its fields. The rest of the pro-
 1186 cess is the same that for *Rose Int*. However, when we encounter the field of type *Rose Int*
 1187 this is already registered, so we just need to use the index *Z* in that position.

1188 The final step is generating the actual Haskell code from the data obtained in the
 1189 previous process. This is a very verbose and mechanical process, whose details we omit.
 1190 In short, we generate the necessary type synonyms, pattern synonyms, the *Family* in-
 1191 stance, and metadata information. The generated type synonyms are named after the
 1192 topmost type of the family, passed to *deriveFamily*:

```
1193 type FamRoseInt = '['Rose Int           , [Rose Int]]
1194 type CodesRoseInt = '['['[K KInt, I (S Z)]] , '['[], '[I Z, I (S Z)]]]
```

1194 The actual *Family* instance is exactly as the one shown in Section 3.1.2

```
1195 instance Family Singl FamRoseInt CodesRoseInt where ...
```

1196 3.1.3.2 METADATA

1197 There is one final ingredient missing to make *generics-mrsop* fully usable in practice.
 1198 We must to maintain the *metadata* information of our datatypes. This metadata includes
 1199 the datatype name, the module where it was defined, and the name of the constructors.
 1200 Without this information we would never be able to pretty print the generic code in a
 1201 satisfactory way. This includes conversion to semi-structured formats, such as JSON, or
 1202 actual pretty printing.

1203 Like in *generics-sop* [23], having the code for a family of datatypes available al-
 1204 lows for a completely separate treatment of metadata. This is yet another advantage
 1205 of the sum-of-products approach compared to the more traditional pattern functors. In

```

data DatatypeInfo :: [[ * ]] → * where
  ADT :: ModuleName → DatatypeName → NP ConstrInfo cs → DatatypeInfo cs
  New  :: ModuleName → DatatypeName → ConstrInfo '[c] → DatatypeInfo '['[c]]

data ConstrInfo :: [ * ] → * where
  Constructor :: ConstrName → ConstrInfo xs
  Infix       :: ConstrName → Associativity → Fixity → ConstrInfo '[x, y]
  Record      :: ConstrName → NP FieldInfo xs → ConstrInfo xs

data FieldInfo :: * → * where
  FieldInfo :: FieldName → FieldInfo a

class HasDatatypeInfo a where
  datatypeInfo :: proxy a → DatatypeInfo (Code a)

```

FIGURE 3.4: Definitions related to metadata from *generics-sop*

fact, our handling of metadata is heavily inspired from *generics-sop*, so much so that we will start by explaining a simplified version of their handling of metadata, and then outline the differences to our approach.

The general idea is to store the meta information following the structure of the datatype itself. Instead of data, we keep track of the names of the different parts and other meta information that can be useful. It is advantageous to keep metadata separate from the generic representation as it would only clutter the definition of generic functionality. This information is tied to a datatype by means of an additional typeclass *HasDatatypeInfo*. Generic functions may now query the metadata by means of functions like *datatypeName*, which reflect the type information into the term level. The definitions are given in Figure 3.4 and follow closely how *generics-sop* handles metadata.

Our library uses the same approach to handle metadata. In fact, the code remains almost unchanged, except for adapting it to the larger universe of datatypes we can now handle. Unlike *generic-sop*, our list of lists representing the sum-of-products structure does not contain types of kind ***, but *Atoms*. All the types representing metadata at the type-level must be updated to reflect this new scenario:

```

data DatatypeInfo :: [[Atom kon]] → * where ...
data ConstrInfo   :: [Atom kon] → * where ...
data FieldInfo    :: Atom kon → * where ...

```

As we have discussed above, our library is able to generate codes not only for single types of kind ***, like *Int* or *Bool*, but also for types which are the result of type-level applications, such as *Rose Int* and *[Rose Int]*. The shape of the metadata information in *DatatypeInfo*, a module name plus a datatype name, is not enough to handle these

cases. We replace the uses of *ModuleName* and *DatatypeName* in *DatatypeInfo* by a richer promoted type *TypeName*, which can describe applications, as required.

```

1227 data TypeName = ConT ModuleName DatatypeName | TypeName :@: TypeName
1228
1229 data DatatypeInfo :: [[Atom kon]] → * where
    ADT :: TypeName → NP ConstrInfo cs → DatatypeInfo cs
    New :: TypeName → ConstrInfo '[c] → DatatypeInfo '[ '[c] ]

```

An important difference to *generics-sop* is that the metadata is not defined for a single type, but for a type *within* a family. This can be seen in the signature of *datatypeInfo*, which receives proxies for both the family and the type. The type equalities in that signature reflect the fact that the given type *ty* is included with index *ix* within the family *fam*. This step is needed to look up the code for the type in the right position of *codes*.

```

1230 class (Family  $\kappa$  fam codes) ⇒ HasDatatypeInfo  $\kappa$  fam codes ix | fam →  $\kappa$  codes where
1231   datatypeInfo :: (ix ~ Idx ty fam , Lkup ix fam ~ ty) ⇒ Proxy fam → Proxy ty
1232   → DatatypeInfo (Lkup ix codes)

```

Template Haskell would generate the instance below for *Rose Int*:

```

1233 instance HasDatatypeInfo Singl FamRose CodesRose Z where
1234   datatypeInfo _ _ = ADT (ConT "E" "Rose" :@: ConT "Prelude" "Int")
1235   $ (Constructor "Fork") ×  $\epsilon$ 

```

3.1.4 EXAMPLE: WELL-TYPED CLASSICAL TREE DIFFERENCING

This section, based on the work of Lempink [51] which originally implemented in the *gdif* library, is the related work that is closest to ours in the sense that it is the only *typed* approach to differencing. The presentation provided here is adapted from Van Putten's [86] master thesis and is available as the *generics-mrsop-gdif* library.

Next, we discuss how to make tree edit-scripts (Section 2.1.2), type-safe following the work of Lempink [51]. We start by lifting edit-scripts to kind $[*] \rightarrow [*] \rightarrow *$, which enables the indexing of the types for the source and destination forests of particular edit-scripts. Consequently, instead of differencing a list of trees, we will difference an *n*-ary product, *NP*, indexed by the type of each tree.

```

1243 type PatchGD  $\kappa$  codes xs ys = ES  $\kappa$  codes xs ys
1244
1245 diff :: (TestEquality  $\kappa$  , EqHO  $\kappa$ )
1246   ⇒ NP (NA  $\kappa$  (Fix  $\kappa$  codes)) xs → NP (NA  $\kappa$  (Fix  $\kappa$  codes)) ys
1247   → PatchGD  $\kappa$  codes xs ys

```

One confusing complication is that our edit operations operate over both constructors of the family and opaque values, unlike the untyped version of tree differencing (Section 2.1.2), where everything is a label. Consequently, writing the edit operations requires a uniform treatment of recursive constructors and opaque values, which is done by the *Cof* type, read as *constructor-of*, and represents the *unit of modification* of each edit operation. A value of type *Cof* κ codes at tys represents a constructor of atom at, which expects arguments whose type is *NP I tys*, for the family codes with opaque types interpreted by κ . Its definition is given below.

```

data Cof  $\kappa$  codes :: Atom kon  $\rightarrow$  [Atom kon]  $\rightarrow$  * where
  ConstrI :: (IsNat c, IsNat n)
     $\Rightarrow$  Constr (Lkup n codes) c  $\rightarrow$  ListPrf (Lkup c (Lkup n codes))
     $\rightarrow$  Cof  $\kappa$  codes ('I n) (Lkup c (Lkup n codes))
  ConstrK ::  $\kappa$  k  $\rightarrow$  Cof  $\kappa$  codes ('K k) Pnil
    
```

We need the *ListPrf* argument to *ConstrI* to be able to manipulate the type-level lists when defining the application function, *applyES*. But first, we have to define our edit-scripts. A value of type *ES* κ codes xs ys represents a transformation of a value of *NP (NA κ (Fix κ codes)) xs* into a value of *NP (NA κ (Fix ki codes)) ys*. The *NP* serves as a list of trees, as is usual for the tree differencing algorithms, but it enables us to keep track of the type of each individual tree through the index to *NP*.

```

data ES  $\kappa$  codes :: [Atom kon]  $\rightarrow$  [Atom kon]  $\rightarrow$  * where
  ESO :: ES  $\kappa$  codes '[]' []
  Ins :: Cof  $\kappa$  codes a t  $\rightarrow$  ES  $\kappa$  codes      i (t : $\#$ : j)  $\rightarrow$  ES  $\kappa$  codes      i (a ': j)
  Del :: Cof  $\kappa$  codes a t  $\rightarrow$  ES  $\kappa$  codes (t : $\#$ : i)      j  $\rightarrow$  ES  $\kappa$  codes (a ': i)      j
  Cpy :: Cof  $\kappa$  codes a t  $\rightarrow$  ES  $\kappa$  codes (t : $\#$ : i) (t : $\#$ : j)  $\rightarrow$  ES  $\kappa$  codes (a ': i) (a ': j)
    
```

Let us take *Ins*, for example. Inserting a constructor *c* :: *t₁ \rightarrow ... \rightarrow t_n \rightarrow 'I ix* in a forest *x₁ \times x₂ \times ... \times Nil* will take the first *n* elements of that forest and use as the arguments to *c*. This is realized by the *insCof* function, shown below.

```

insCof :: Cof  $\kappa$  codes a t
   $\rightarrow$  NP (NA  $\kappa$  (Fix  $\kappa$  codes)) (t : $\#$ : xs)  $\rightarrow$  NP (NA  $\kappa$  (Fix  $\kappa$  codes)) (a ': xs)
insCof (ConstrK k)      xs = NAK k  $\times$  xs
insCof (ConstrI c ispoa) xs = let (poa, xs') = split ispoa xs in NAI (Fix $ inj c poa)  $\times$  xs'
    
```

The example also showcases the use of the *ListPrf* present in *ConstrI*, which is necessary to enable us to split the list *t : $\#$: xs* into *t* and *xs*. The typechecker needs some more information about *t*, since type families are not injective. The *split* function has type:

```

split :: ListPrf xs  $\rightarrow$  NP p (xs : $\#$ : ys)  $\rightarrow$  (NP p xs, NP p ys)
    
```

1273 The *delCof* function is dual to *insCof*, but since we construct a *NP* indexes over $t : \# :$
 1274 *xs*, we need not use the *ListPrf* argument. Finally, we can assemble the application
 1275 function that witnesses the semantics of *ES*:

$$\begin{aligned} \text{applyES} &:: (\forall k . \text{Eq } (\kappa k)) \Rightarrow \text{ES } \kappa \text{ codes } xs \text{ } ys \rightarrow \text{PoA } \kappa (\text{Fix } \kappa \text{ codes}) xs \\ &\quad \rightarrow \text{Maybe } (\text{PoA } \kappa (\text{Fix } \kappa \text{ codes}) ys) \\ \text{applyES ES0} &\quad \quad \quad = \text{Just Nil} \\ \text{applyES (Ins } _ \text{ c es) } xs &= \text{insCof c } \langle \$ \rangle \text{ applyES es } xs \\ \text{applyES (Del } _ \text{ c es) } xs &= \text{delCof c } xs \succcurlyeq \text{ applyES es} \\ \text{applyES (Cpy } _ \text{ c es) } xs &= \text{insCof c } \langle \$ \rangle (\text{delCof c } xs \succcurlyeq \text{ applyES es}) \end{aligned}$$

1277 3.1.4.1 DISCUSSION

1278 The approach of providing typed edit operations has many nice aspects. It immediately
 1279 borrows the existing algorithms and metatheory and can improve the size of edit-scripts
 1280 significantly by being able to provide *CpyTree*, *InsTree* and *DelTree* which copy, insert
 1281 and delete entire trees instead of operating on individual constructors. This is possible
 1282 because we can look at the type of the edit-script in question – substitute the insertion of
 1283 a constructor by *InsTree* whenever all of its fields are also comprised solely of insertions.

1284 Although type-safe by construction, which is undoubtedly a plus point, computing
 1285 edit-scripts, with memoization, still takes $\mathcal{O}(n \times m)$ time, where n and m are the number
 1286 of constructors in the source and destination trees. This means this is at least quadratic
 1287 in the size of the smaller input, which is not practical for a tool that is supposed to be
 1288 run multiple times per commit on large inputs. This downside is not specific to this
 1289 approach, but rather quite common for tree differencing algorithms. They often belong
 1290 to complexity classes that make them impractical.

1291 Another downside comes to the surface when we want to look into merging these
 1292 edit-scripts. Vassena [100] developed a merging algorithm but notes some difficult set-
 1293 backs, mainly due to the heterogeneity of *ES*. Suppose, for example, we want to merge
 1294 $p : \text{ES } xs \text{ } ys$ and $q : \text{ES } xs \text{ } zs$. This means producing an edit-script $r : \text{ES } xs \text{ } ks$. But
 1295 how can we determine ks here? It is not always the case that there is a solution. In fact,
 1296 the merge algorithm [100] for *ES* might fail due to conflicting changes or the inability to
 1297 find a suitable ks . Regardless, the work of Vassena [100] was of great inspiration for this
 1298 thesis in showing that there definitely is a place for type-safe approaches to differencing.

1299 3.2 THE GENERICS-SIMPLISTIC LIBRARY

1300 Unfortunately, the *generics-mrsop* uncovered a memory leak in the Haskell compiler
 1301 itself when used for large mutually recursive families. The bugs have been reported in

1302 the GHC bug tracker² but at the time of writing of this thesis, have not been resolved.
 1303 This means that if we wish to collect large scale real data for our experiments, we must
 1304 develop and alternative approach.

1305 ◁ *After realizing that the differencing algorithms presented in Chapter 5 did not ex-*
 1306 *PLICITLY require sums of products to work, I was able to implement a workaround using*
 1307 *GHC.Generics to encode mutually recursive families. The main idea is to take the dual*
 1308 *approach from generics-mrsop: instead of defining which types belong in the family,*
 1309 *we define which types do not belong to the family. Corresponding with A. Serrano we dis-*
 1310 *cussed how this approach could be seen as an extension of his generics-simplistic*
 1311 *library, which lead me to write the layer that handles deep representations with support*
 1312 *for mutual recursion on top a the preliminary version of this library, giving rise to the*
 1313 *generics-simplistic library in its current form.* ▷

1314 3.2.1 THE SIMPLISTIC VIEW

1315 The generics-simplistic library can be seen as a layer on top of GHC.Generics to
 1316 ease out the definition of new generic functionality. The pattern functor approach used
 1317 by GHC.Generics, shown in Section 2.2.1, requires the user to write a large number
 1318 of typeclass instances to define even basic generic functions. Yet, the pattern functors
 1319 generated by GHC are restricted to sums, products, unit, constants and metadata in-
 1320 formation. This means we can model representations as a single GADT, *SRep* defined
 1321 below, indexed by the pattern functor it inhabits.

```

1322 data SRep (φ :: * → *) :: (* → *) → * where
    S_U1 :: SRep φ U1
    S_K1 :: φ a → SRep φ (K1 i a)
    S_L1 :: SRep φ f → SRep φ (f :+: g)
    S_R1 :: SRep φ g → SRep φ (f :+: g)
    (:*:) :: SRep φ f → SRep φ g → SRep φ (f :* g)
    S_M1 :: SMeta i t → SRep φ f → SRep φ (M1 i t f)
    
```

1323 The handling of metadata is borrowed entirely from GHC.Generics and captured
 1324 by the *SMeta* datatype, which records the kind of meta-information is stored at the type-
 1325 level.

```

1326 data SMeta i t where
    SM_D :: Datatype d ⇒ SMeta D d
    SM_C :: Constructor c ⇒ SMeta C c
    SM_S :: Selector s ⇒ SMeta S s
    
```

² <https://gitlab.haskell.org/ghc/ghc/issues/17223> and <https://gitlab.haskell.org/ghc/ghc/issues/14987>

1327 The *SRep* datatype enables us to write generic functionality more concisely than
 1328 `GHC.Generics`. Take the *gsi* function from Section 2.2.1 as an example. With pure
 1329 `GHC.Generics`, we must use *Size* and *GSize* typeclasses. With *SRep* we can write it
 1330 directly, provided we have a way to count the size of the leaves of type φ .

```

gsi :: (forall x . phi x -> Int) -> SRep phi f -> Int
gsi r S_U1      = 0
gsi r (S_K1     x) = r x
1331 gsi r (S_M1    _ x) = gsi r x
gsi r (S_L1     x) = gsi r x
gsi r (S_R1     x) = gsi r x
gsi r (x :: y)   = gsi r x + gsi r y

```

1332 Naturally, we still need to convert values of *GHC.Generics.Rep* f x into their closed
 1333 representation, *SRep* φ (*GHC.Generics.Rep* f) and make some choice for φ . We could
 1334 use *K1 R* as φ , essentially translating only the first layer into a generic representation,
 1335 but as we shall see in Section 3.2.2, we can also translate the entire value and uses a
 1336 fixpoint combinator in φ .

1337 Even though *SRep* lacks a *codes-based* approach, that is, it can be defined for arbitrary
 1338 types like `GHC.Generics`, it still admits some combinators that greatly assist a
 1339 programmer when writing their generic code, unlike `GHC.Generics`. The most useful
 1340 being *repMap*, *repZip* and *repLeaves*, that map, zip and collect the leaves of a *SRep*
 1341 respectively. These can easily be generalized to a monadic version.

```

repMap :: (forall x . phi x -> psi x) -> SRep phi f -> SRep psi f
1342 repZip :: SRep phi f -> SRep psi f -> Maybe (SRep (phi :: psi) f)
repLeaves :: SRep phi f -> [Exists phi]

```

1343 3.2.2 MUTUAL RECURSION

1344 The *SRep* φ f datatype enables us to write generic functions without resorting to type-
 1345 classes and also provides a simple way to interact with potentially recursive subtrees
 1346 through the φ functor. To write a deep representation, all we have to do is define a mu-
 1347 tually recursive family to be any type that is *not* a primitive type, where the choice of
 1348 primitive type shall be parameterize through the usual κ parameter. The pseudo-code
 1349 below illustrates this idea.

```

data SFix kappa :: * -> * where
1350 Prim :: (x in kappa) => x -> SFix kappa fam x
      SFix :: (not (x in kappa), Generic x) => SRep (SFix prim) (Rep x) -> SFix kappa fam x

```


This approach works well for simpler applications, but by defining a mutually recursive family in an *open* fashion, i.e., t is an element iff $\neg (t \in \kappa)$, for some list κ of types regarded as primitive, we would only be able to check for index equality through the *Typeable* machinery [43], which would have to spread across the library, inherently breaking parametricity of maps and catamorphisms besides polluting the interface. Checking for index equality is crucial for the definition of many generic concepts – zippers being a prominent example, Section 3.2.4.1 – and was trivial to define in `generics-mrsop`, thanks to its *closed* approach: if two types were identified by the same index into a list containing all members of the family, then they are the same type.

To avoid having to spread *Typeables* around but still maintaining decidable type index equality we will apply the same trick here: define a family as two disjoint lists: A type-level list *fam* for the elements that belong in the family and one for the primitive types, usually denoted κ . Note that unlike `generics-mrsop`, κ here has kind $'[*]$.

Recursion is easily achieved through a *SFix* κ *fam* combinator, where *fam* $:: '[*]$ is the list of types that belong in the family and $\kappa :: '[*]$ is the list of types to be considered primitive, that is, is not unfolded into a generic representation. The *SFix* combinator has two constructors, one for carrying values of primitive types and one for unfolding a next layer of the generic representation, as defined below.

```
data SFix  $\kappa$  fam  $:: * \rightarrow *$  where
  Prim  $:: (PrimCnstr\ \kappa\ fam\ x) \Rightarrow x \rightarrow SFix\ \kappa\ fam\ x$ 
  SFix  $:: (CompoundCnstr\ \kappa\ fam\ x) \Rightarrow SRep\ (SFix\ prim)\ (Rep\ x) \rightarrow SFix\ \kappa\ fam\ x$ 
```

Here, *PrimCnstr* and *CompoundCnstr* are constraint synonyms, defined below, to encapsulate what it means for a type x to be primitive (resp. compound) with respect to the *fam* and *prim* list of types.

```
type PrimCnstr       $\kappa\ fam\ x = (Elem\ x\ \kappa,\ NotElem\ x\ fam)$ 
type CompoundCnstr  $\kappa\ fam\ x = (Elem\ x\ fam,\ NotElem\ x\ \kappa,\ Generic\ x)$ 
```

Elem and *NotElem* are custom constraints that state whether or not a type is an element of a list of types. They are defined with the help of the boolean type family and, in the *Elem* case, we also carry a typeclass that enables us to construct a membership proof.

```
type Elem       $a\ as = (IsElem\ a\ as \sim 'True,\ HasElem\ a\ as)$ 
type NotElem  $a\ as = IsElem\ a\ as \sim 'False$ 
type family IsElem  $(a :: *) (as :: [*]) :: Bool$  where
  IsElem  $a\ [] = 'False$ 
  IsElem  $a\ (a' :: as) = 'True$ 
  IsElem  $a\ (b' :: as) = IsElem\ a\ as$ 
```

1379 *HasElem* a as , here, is a typeclass that produces an actual proof that the list as
 1380 contains a – encoded in a datatype *ElemPrf* a as . Pattern matching on a value of type
 1381 *ElemPrf* a as will unfold the structure of as . This is crucial in, for example, access-
 1382 ing typeclass instances for types in *SFix* κ fam . The *HasElem* typeclass and *ElemPrf*
 1383 datatype are defined below.

```

1384 data ElemPrf a as where
    Here :: ElemPrf a (a ' : as)
    There :: ElemPrf a as → ElemPrf a (b ' : as)

class HasElem a as where
    hasElem :: ElemPrf a as
  
```

1385 To define generic functions, we often need operation over the primitive types. We
 1386 can encode this via constraints, requiring that *all* elements of κ have instances of some
 1387 typeclass. Suppose we would like to write a term-level equality operator for values of
 1388 type *SFix* κ fam x , as in the *Eq* typeclass. This would require to ultimately compare
 1389 values of type y , for some y such that *Elem* y κ . Naturally, this can only be done if all
 1390 elements of κ are members of the *Eq* typeclass. We specify that all elements of κ satisfy
 1391 a constraint with the *All* [23] type family:

```

1392 type family All c xs :: Constraint where
    All c '[] = ()
    All c (x (' : xs) = (c x, All c xs)
  
```

1393 Now, given a function with type $(All\ Eq\ prim) \Rightarrow SFix\ prim\ x \rightarrow \dots$, we must
 1394 extract the *Eq* y instance from *All* *Eq* $prim$, for some y such that *IsElem* y $prim \sim 'True$.
 1395 This is where *ElemPrf* becomes essential. By pattern matching on *ElemPrf* we are able to
 1396 extract the necessary instance, as shown by the *witness* function below. Naturally, once
 1397 we find the instance we are looking for, we record it in a datatype for easier access.

```

1398 data Witness c x where
    Witness :: (c x) ⇒ Witness c x

witness :: ∀ x xs c . (HasElem x xs, All c xs) ⇒ Proxy xs → Witness c x
witness _ = witnessPrf (hasElem :: ElemPrf x xs)
    where witnessPrf :: (All c xs) ⇒ ElemPrf x xs → Witness c x
          witnessPrf Here = Witness
          witnessPrf (There p) = witnessPrf p
  
```

1399 The *witness* function above enables us to cast the usual (\equiv) function, from *Eq*, as
 1400 operating over any element of a list of types. Pattern matching on the result of *witness*
 1401 enables the compiler to access the necessary *Eq* instance. With the help of *weg* below,
 1402 we define the *Eq* instance for *SFix* in Figure 3.5. Note that calling *witness* will require

```

instance (All Eq  $\kappa$ )  $\Rightarrow$  Eq (SFix  $\kappa$  fam f) where
  (Prim x)  $\equiv$  (Prim y) = weq x y
  (SFix x)  $\equiv$  (SFix y) = maybe False (all ( $\equiv$ )  $\circ$  repLeaves) (repZip x y)

```

FIGURE 3.5: Equality instance for *SFix*.

1403 an explicit type annotation informing the compiler about which typeclass we wish to
 1404 extract from the top-level *All* constraint.

```

1405 weq ::  $\forall$  x xs . (All Eq xs, Elem x xs)  $\Rightarrow$  Proxy xs  $\rightarrow$  x  $\rightarrow$  x  $\rightarrow$  Bool
  weq p = case witness p :: Witness Eq x of Witness  $\rightarrow$  ( $\equiv$ )

```

1406 With the *Elem* functionality in place, we can define type-level equality for elements
 1407 of a given list – given *SFix* κ fam x and *SFix* κ fam y, to be able to know whether $x \sim y$.
 1408 This functionality is important when defining the zipper [41] or generic unification, and
 1409 it comes for free in code-based approaches, such as *generics-mrsop*. In our current
 1410 setting, we need to use the *fam* type-level list and the *HasElem* typeclass. Note that the
 1411 proxies are present solely to aid the reduction of the *IsElem* type family, needed for *Elem*.

```

  sameType :: (Elem x fam, Elem y fam)
     $\Rightarrow$  Proxy fam  $\rightarrow$  Proxy x  $\rightarrow$  Proxy y  $\rightarrow$  Maybe (x  $\sim$  y)
  sameType _ _ = sameIdx (hasElem :: ElemPrf x fam) (hasElem :: ElemPrf y fam)
1412 where sameIdx :: ElemPrf x xs  $\rightarrow$  ElemPrf x' xs  $\rightarrow$  Maybe (x  $\sim$  x')
    sameIdx Here Here = Just Refl
    sameIdx (There rr) (There y) = go rr y
    sameIdx _ _ = Nothing

```

1413 CONVERTING TO A DEEP REPRESENTATION. With representational issues out of the
 1414 way, we shall need to translate between a value and its deep *GHC.Generics*-based rep-
 1415 resentation. This can be done with the generic functions *dfrom* and *dto*, which follow
 1416 the textbook recipe of defining generic functionality with *GHC.Generics*: use a type-
 1417 class and its generic variant and use *default signatures* to bridge the gap between them.
 1418 In our case, this is done with the *Deep* and *GDeep* typeclasses, declared in Figure 3.6.

1419 Defining the *GDeep* instances is straightforward with the exception of the (*K1 R a*)
 1420 case, where we must decide whether or not *a* is a primitive type. Ideally we would like
 1421 to write something in the lines of:

```

1422 instance (IsElem a  $\kappa \sim$  'True')  $\Rightarrow$  GDeep  $\kappa$  fam (K1 R a) ...
instance (IsElem a  $\kappa \sim$  'False')  $\Rightarrow$  GDeep  $\kappa$  fam (K1 R a) ...

```

```

class (CompoundCnstr  $\kappa$  fam a)  $\Rightarrow$  Deep  $\kappa$  fam a where
  dfrom :: a  $\rightarrow$  SFix  $\kappa$  fam a
  default dfrom :: (GDeep  $\kappa$  fam (Rep a))  $\Rightarrow$  a  $\rightarrow$  SFix  $\kappa$  fam a
  dfrom = SFix  $\circ$  gdfrom  $\circ$  from
  dto :: SFix  $\kappa$  fam a  $\rightarrow$  a
  default dto :: (GDeep  $\kappa$  fam (Rep a))  $\Rightarrow$  SFix  $\kappa$  fam a  $\rightarrow$  a
  dto (SFix x) = to (gdto x)

class GDeep  $\kappa$  fam f where
  gdfrom :: f x  $\rightarrow$  SRep (SFix  $\kappa$  fam) f
  gdto    :: SRep (SFix  $\kappa$  fam) f  $\rightarrow$  f x

```

FIGURE 3.6: Declaration of *Deep* and *GDeep* typeclasses

1423 But GHC cannot distinguish between these two instances when resolving them. Not
 1424 even `-XOverlappingInstances` can help us here. The only way out is to abstract the
 1425 call to *IsElem* to an auxiliary typeclass, which “pattern matches” on the result of this
 1426 type-level computation.

```

class GDeepAtom  $\kappa$  fam (isPrim :: Bool) a where
  1427   gdfromAtom :: Proxy isPrim  $\rightarrow$  a  $\rightarrow$  SFix  $\kappa$  fam a
   gdtoAtom    :: Proxy isPrim  $\rightarrow$  SFix  $\kappa$  fam a  $\rightarrow$  a

```

1428 The *GDeepAtom* class possesses only two instances, one for primitive types and one
 1429 for types we wish to consider as members of our mutually recursive family, which are
 1430 indicated by the *isPrim* parameter. We recall the definitions for *CompoundCnstr* and
 1431 *PrimCnstr* below.

```

instance (CompoundCnstr  $\kappa$  fam a, Deep  $\kappa$  fam a)  $\Rightarrow$  GDeepAtom  $\kappa$  fam 'False a ...
instance (PrimCnstr       $\kappa$  fam a)                 $\Rightarrow$  GDeepAtom  $\kappa$  fam 'True a ...
  1432
type PrimCnstr       $\kappa$  fam x = (Elem x  $\kappa$ , NotElem x fam)
type CompoundCnstr  $\kappa$  fam x = (Elem x fam, NotElem x  $\kappa$ , Generic x)

```

1433 Finally, the actual instance for *GDeep prim (K1 R a)* triggers the evaluation of
 1434 *IsElem*, which in turn brings into scope the correct variation of the *GDeepAtom*:

```

  1435 instance (GDeepAtom  $\kappa$  fam (IsElem a prim) a)  $\Rightarrow$  GDeep  $\kappa$  fam (K1 R a) where

```

1436 With the *Deep* typeclass setup, all we have to do is declare an empty instance for
 1437 every element of the family. Figure 3.7 illustrates the usage for the *Rose* datatype. The
 1438 monomorphic versions of *dfrom* and *dto* simply aid the compiler by providing all neces-
 1439 sary type parameters.

```

data Rose a = Fork a [Rose a]
    deriving (Eq, Show, Generic)
type RosePrims = '[Int]
type RoseFam  = '[Rose Int, [Rose Int]]
instance Deep RosePrims RoseFam (Rose Int)
instance Deep RosePrims RoseFam [Rose Int]
dfromRose :: Rose Int → SFix RosePrims RoseFam (Rose Int)
dfromRose = dfrom
dtoRose :: SFix RosePrims RoseFam (Rose Int) → Rose Int
dtoRose = dto

```

FIGURE 3.7: Usage example for *generics-simplistic*

3.2.3 THE (CO)FREE (CO)MONAD

Although the *SFix* type makes for a very intuitive recursion combinator, it does not give us much flexibility: it does not support annotations nor holes. For example, suppose we want to define a generic unification algorithm: how would we represent unification variables within *SFix*? We would an augmented *SFix* which would carry one extra constructor for unification variables. Another example would be annotating an *SFix* with some auxiliary values to make certain computations more efficient. These variations over fixpoints can be achieved by combining the free monad and the cofree comonad in the same type, which we name *HolesAnn* κ fam φ *h* *a*. A value of type *HolesAnn* κ fam φ *h* *a* is isomorphic to a value of type *a*, where each constructor is annotated with φ and we might have holes of type *h*.

```

data HolesAnn  $\kappa$  fam  $\varphi$  h a where
    Hole' ::  $\varphi$  a → h a → HolesAnn  $\kappa$  fam  $\varphi$  h a
    Prim' :: (PrimCnstr  $\kappa$  fam a) ⇒  $\varphi$  a → a → HolesAnn  $\kappa$  fam  $\varphi$  h a
    Roll'  :: (CompoundCnstr  $\kappa$  fam a) ⇒  $\varphi$  a → SRep (HolesAnn  $\kappa$  fam  $\varphi$  h) (Rep a)
        → HolesAnn  $\kappa$  fam  $\varphi$  h a

```

The *SFix* combinator presented earlier can be easily seen as the special case where annotations are the unit type, *U1*, and holes do not exist (which is captured by the empty type *V1*). We can represent all the variations over fixpoints through type synonyms:

```

type SFix       $\kappa$  fam  = HolesAnn  $\kappa$  fam U1 V1
type SFixAnn  $\kappa$  fam  $\varphi$  = HolesAnn  $\kappa$  fam  $\varphi$  V1
type Holes      $\kappa$  fam  = HolesAnn  $\kappa$  fam U1

```

Again, with the help of pattern synonyms and COMPLETE pragmas – which stops GHC from issuing `-Wincomplete-patterns` warnings – we can simulate the `SFixAnn` datatype, for example.

```

pattern SFixAnn :: () => (CompoundCnstr κ fam a)
                    => φ a → SRep (SFixAnn κ fam φ) (Rep a) → SFixAnn κ fam φ a
pattern SFixAnn ann x = Roll' ann x
pattern PrimAnn :: () => (PrimCnstr κ fam a) => φ a → a → SFixAnn κ fam ann a
pattern PrimAnn ann x = Prim' ann x
{-# COMPLETE SFixAnn , PrimAnn #-}

```

Annotated fixpoints, in fact, are very important for us. Many of the algorithms in Chapter 5 proceed by first annotating a tree with some auxiliary information and then computing a result over said tree. This ensures we never recompute auxiliary information and keeps our algorithms linear.

3.2.3.1 ANNOTATED FIXPOINTS

Catamorphisms are used in a large number of computations over recursive structures. They receive an algebra that is used to consume one layer of a datatype at a time and consumes the whole value of the datatype using this *recipe*. The definition of the catamorphism is trivial in a setting where we have explicit recursion:

```

cata :: (∀ b . (CompoundCnstr κ fam b) => SRep φ (Rep b) → φ b)
      → (∀ b . (PrimCnstr κ fam b) => b → φ b) → SFix κ fam h a → φ a
cata f g (SFix x) = f (repMap (cata f g) x)
cata _ g (Prim x) = g x

```

One example of catamorphisms is computing the *height* of a recursive structure. It can be defined with `cata` in a simple manner with the help of the `Const` functor.

```

newtype Const t x = Const {getConst :: t}
heightAlgebra :: SRep (Const Int) xs → Const Int iy
heightAlgebra = Const ∘ (1+) ∘ maximum ∘ (0:) ∘ map (exElim getConst) ∘ repLeaves
height :: SFix κ fam a → Int
height = getConst ∘ cata heightAlgebra

```

Now imagine our particular application makes a number of decisions based on the height of the (generic) trees it handles. Calling *height* at each of those decision points is time consuming. It is much better to compute the height of a tree only once and keep the intermediary results annotated in their respective subtrees. We can easily do so with

our *SFixAnn* *cofree comonad* [34], in fact, we would say that the height is a synthesized attribute in *attribute grammar* [48] lingo.

```

1477   synthesize :: (∀ b . (CompoundCnstr κ fam a) ⇒ SRep φ (Rep b) → φ b)
1478               → (∀ b . (PrimCnstr κ fam a) ⇒ b → φ b)
1479               → SFix κ fam a → SFixAnn κ fam φ a
    synthesize f g = cata (λr → SFixAnn (f (repMap getAnn r)) r) (λa → PrimAnn (g b) b)

```

Finally, an algorithm that constantly queries the height of the subtrees can be computed in two passes: in the first pass we compute the heights and leave them annotated in the tree, in the second we run the algorithm. Moreover, we can compute all the necessary synthesized attributes an algorithm needs in a single preprocessing phase. This is a crucial maneuver to make sure our generic programs can scale to real-world inputs. Naturally, *cata* and *synthesize* are actually implemented in their monadic form and over *HolesAnn* for maximal generality.

3.2.4 PRACTICAL FEATURES

Whilst developing *hdiff* (Chapter 5), we ran into a number of practicalities regarding the underlying generic programming library. Of particular importance are zippers and unification, which play a big role in the algorithms underlying the *hdiff* approach. This section gives an overview of those features.

3.2.4.1 ZIPPERS

Zippers [41] are a well established technique for traversing a recursive data structure keeping track of a focus point. Defining generic zippers is not new, this has been done by many authors [1, 40, 105] for many different classes of datatypes in the past. In our particular case, we are not interested in traversing a generic representation by means of the usual zipper traversals – up, down, left and right – which move the focus point. Instead, we just want a datatype that encodes a context with one focus, encoded by *SZip* below. A value of type *SZip* *ty w f* represents a value of type *SRep* *w f* with one *hole*, or *focus*, in a position with type *ty*.

```

1492   data SZip ty w f where
1493       Z_L1  :: SZip ty          w f → SZip ty w (f :+: g)
1494       Z_R1  :: SZip ty          w g → SZip ty w (f :+: g)
1495       Z_PairL :: SZip ty w f → SRep w g → SZip ty w (f **: g)
1496       Z_PairR :: SRep w f → SZip ty w g → SZip ty w (f **: g)
1497       Z_M1   :: SMeta i t → SZip ty w f → SZip ty w (M1 i t f)
1498       Z_KH   ::              → SZip ty w (K1 i a)

```

1502 The *Zipper* datatype will ensure that the focus lies in a recursive position. Its defi-
 1503 nition is given below. It encapsulates the *ty* above as an existential type and keeps the
 1504 focus point accessible. We also pass around a constraint-kinded variable to enable one
 1505 to specify custom constraints about the types in question.

```
1506 data Zipper c f g t where
      Zipper :: c => SZip t f (Rep t) -> g t -> Zipper c f g t
```

1507 Given a value of type *SZip ty φ t* and a value of type *φ ty*, it is straightforward to
 1508 plug the hole and produce a *SRep φ t*. The other way around, however, is more com-
 1509 plicated. Given a *SRep φ t*, we might have many possible zippers – binary trees, for
 1510 example, can have a hole on the left or on the right branch. Consequently, we must re-
 1511 turn a list of zippers. The *zippers* function below does exactly that. Its type is convoluted
 1512 because it works over *HolesAnn* (and therefore also for *SFix*, *SFixAnn* and *Holes*), but
 1513 it is conceptually simple: given a test for whether a hole of type *φ a* is actually a hole
 1514 in a recursive position, we return the list of possible zippers for a value with holes. The
 1515 definition is standard and we encourage the interested reader to check the source code
 1516 for more details, Appendix A.

```
type Zipper' κ fam φ h t
  = Zipper (CompoundCnstr κ fam t) (HolesPhi κ fam φ h) (HolesPhi κ fam φ h) t
zippers :: (∀ a . (Elem t fam) => h a -> Maybe (a ~: t))
  -> HolesPhi κ fam φ h t -> [Zipper' κ fam φ h t]
```

1518 3.2.4.2 UNIFICATION AND ANTI-UNIFICATION

1519 Both unification and anti-unification algorithms make up an important part of the ver-
 1520 nacular of term-manipulation. Unsurprisingly, we will also need to implement these
 1521 features into *generics-simplistic*. We use them extensively in Chapter 5. This sec-
 1522 tion provides an overview of the (anti-)unification provided by *generics-simplistic*.

1523 Syntactic unification algorithms [89] receive as input two terms *t* and *u* with vari-
 1524 ables and outputs substitutions *σ* such that *σ t* \equiv *σ u*, when such *σ* exists. Anti-
 1525 unification[85], on the other hand, receives two terms *t* and *u* and outputs one term
 1526 *r* and two substitutions *σ* and *φ* such that *t* \equiv *σ r* and *u* = *φ r*.

1527 With our current setup, we want to unify two terms of type *Holes κ fam φ at*, that
 1528 is, two elements of the mutually recursive family *fam* with unification variables of type
 1529 *φ*. A substitution is given by:

```
1530 type Subst κ fam φ = Map (Exists φ) (Exists (Holes κ fam φ))
```


1531 We need the existentials here in order to use the builtin, homogeneous, *Data.Map*.
 1532 \triangleleft we could write a custom heterogeneous key-value store, but I'm doubtful this would be
 1533 worth the trouble. *Data.Map* has excellent performance and has been thoroughly tested.
 1534 \triangleright Naturally, when looking for the value associated with a key within the substitution
 1535 we will run into a type error as soon as we unwrap the *Exists*. There are a number of
 1536 solutions to this. For one, we could use the *sameTy* function and ensure they are of
 1537 the same type. Pragmatically though, as long as we ensure we only insert keys φ at
 1538 associated with values *Holes* κ fam φ at, the type indexes will never differ and we
 1539 can safely call *unsafeCoerce* to mitigate any performance overhead. We chose to use
 1540 *unsafeCoerce* but stress that it can be easily avoided with a call to *sameTy*.

```

1541      substInsert :: (Ord (Exists  $\varphi$ ))  $\Rightarrow$  Subst  $\kappa$  fam  $\varphi \rightarrow \varphi$  at  $\rightarrow$  Holes  $\kappa$  fam  $\varphi$  at
       $\rightarrow$  Subst  $\kappa$  fam  $\varphi$ 
      substLkup :: (Ord (Exists  $\varphi$ ))  $\Rightarrow$  Subst  $\kappa$  fam  $\varphi \rightarrow \varphi$  at  $\rightarrow$  Maybe (Holes  $\kappa$  fam  $\varphi$  at)
    
```

1542 When attempting to solve a unification problem, there are two types of failures that
 1543 can occur: symbol clashes happen when we try to unify different symbols, for example,
 1544 $c\ x$ is not unifiable with $c'\ x$ because $c \not\equiv c'$; and occurs check errors are raised when
 1545 there is a loop in the substitution, for example, if we try to unify $c\ (c'\ x)$ with $c\ x$, we
 1546 would have to substitute x for $c'\ x$, but this would never terminate. We encode these
 1547 errors in the *UnifyErr* datatype, making it easy for users of the library to catch these
 1548 errors and extract information from them.

```

1549      data UnifyErr  $\kappa$  fam  $\varphi$  where
      OccursCheck :: [Exists  $\varphi$ ]  $\rightarrow$  UnifyErr  $\kappa$  fam  $\varphi$ 
      SymbolClash :: Holes  $\kappa$  fam  $\varphi$  at  $\rightarrow$  Holes  $\kappa$  fam  $\varphi$  at  $\rightarrow$  UnifyErr  $\kappa$  fam  $\varphi$ 
    
```

1550 The *unify* function has the type one would expect: given two terms with unification
 1551 variables of type φ , either they are not unifiable or there exists a substitution that makes
 1552 them equal.

```

1553      unify :: (Ord (Exists  $\varphi$ ), EqHO  $\varphi$ )  $\Rightarrow$  Holes  $\kappa$  fam  $\varphi$  at  $\rightarrow$  Holes  $\kappa$  fam  $\varphi$  at
       $\rightarrow$  Except (UnifyErr  $\kappa$  fam  $\varphi$ ) (Subst  $\kappa$  fam  $\varphi$ )
    
```

1554 Our *unify* function is a constraint-based unifier which computes the most general
 1555 unifier in two phases: first it collects all the necessary equivalences, then it tries to pro-
 1556 duce an idempotent substitution from the gathered equivalences. We omit technical
 1557 details regarding the implementation of the unification algorithm and refer the reader
 1558 to the existing literature [89].

1559 Anti-unification [85] is dual to unification. It is the process of identifying the the
 1560 longest prefixes that two terms agree. For example, take $x = \text{Bin } (\text{Bin } 1\ 2)\ \text{Leaf}$ and
 1561 $y = \text{Bin } (\text{Bin } 1\ 3)\ (\text{Bin } 4\ 5)$, the term $\text{Bin } (\text{Bin } 1\ a)\ b$ is the least general generalization

```

lgg :: (All Eq κ) ⇒ Holes κ fam φ at → Holes κ fam ψ at
    → Holes κ fam (Holes κ fam φ :: Holes κ fam ψ) at
lgg (Prim x) (Prim y) =
  | weq (Proxy :: Proxy κ) x y = Prim x
  | otherwise                    = (Prim x :: Prim y)
lgg x@(Roll rx) y@(Roll ry) = case zipSRep rx ry of
  Nothing → Hole (x :: y)
  Just r  → Roll (repMap (uncurry' lgg) r)
lgg x y = Hole (x :: y)

```

FIGURE 3.8: Classic anti-unification algorithm [85], producing the least general generalization of two trees.

of x and y . That is, there exist two instantiations of a and b yielding x or y . The term $\text{Bin } c \ b$ is also a generalization of x and y , but it is not the *least* general because to obtain x or y we would have to instantiate c as $\text{Bin } 1 \ 2$ or $\text{Bin } 1 \ 3$, and these terms can be further anti-unified. Figure 3.8 illustrates the implementation of the syntactical anti-unification algorithm.

3.3 DISCUSSION

In this chapter we explored two different ways of writing generic programs that must work over mutually recursive families. Looking back at the spectrum of generic programming libraries, in Figure 2.10, we had a unfilled hole for *code-based* approach with explicit recursion of any type, which can be filled by `generics-mrsop`. When it comes to pattern functors, although `regular` and `multirec` already exist, using those libraries imposes a significant overhead when compared to `generics-simplistic`, for they do not support combinator-based generic programming. The updated table of generic programming libraries is given in Figure 3.9, where we place our libraries in the spectrum of generic programming variants.

Unfortunately, the `generics-mrsop` heavy usage of type families triggers a memory leak in the compiler. This renders the library unusable for large families of mutually recursive datatypes at the time of writing this thesis. Luckily, however, we were able to work around that by dropping the sums of products structure but maintaining a combinator-based approach in `generics-simplistic`, which enabled us to run our experiments with real-world data, as discussed in Chapter 6.

While developing the `generics-mrsop` and `generics-simplistic` libraries, which happened under close collaboration with Alejandro Serrano, we also explored a number of variants of these libraries such as `kind-generics` [93], which enables a

	Pattern Functors	Codes
No Explicit Recursion	<code>GHC.Generics</code>	<code>generics-sop</code>
Simple Recursion	<code>generics-simplistic</code>	<code>generics-mrsop</code>
Mutual Recursion		

FIGURE 3.9: *Updated spectrum of generic programming libraries*

1586 user to represent almost any Haskell datatype generically, including *GADTs*. These are
 1587 out of the scope of this thesis since we do not require all of that expressivity to write our
 1588 differencing algorithms.

1589



1590

1591

STRUCTURAL PATCHES

1592 The `gdiff` [51] approach, discussed in Section 3.1.4, which flattens a tree into a list,
 1593 following classical tree edit distance algorithms encoded through using type-safe edit-
 1594 scripts, inherits the problems of edit-script based approaches. These include ambiguity
 1595 on the representation of patches, non-uniqueness of optimal solutions and difficulty of
 1596 merging. The `stdiff` approach, discussed through this chapter, arose from our study
 1597 of the difficulties about merging `gdiff` patches [100].

1598 The heterogeneity of Patch_{GD} makes merging difficult. Recall that a value of type
 1599 $\text{Patch}_{\text{GD}}\ x\ y$ transforms a list of trees x into a list of trees y . If we are given two patches
 1600 $\text{Patch}_{\text{GD}}\ x\ y$ and $\text{Patch}_{\text{GD}}\ x\ z$, we would like to produce two patches $\text{Patch}_{\text{GD}}\ y\ r$
 1601 and $\text{Patch}_{\text{GD}}\ z\ r$ such that the canonical merge square commutes. The problem be-
 1602 comes clear when we try to determine r correctly: sometimes such r might not even
 1603 exist [100].

1604 Our `stdiff` approach, or, *structural patches*, marks our first attempt at defining a
 1605 *type-indexes* patch datatype, Patch_{ST} , in pursuit of better behaved merge algorithms. The
 1606 overall idea consists in making sure that the type of patches is also *tree structured*, as
 1607 opposed to managing a list-like patch data structure that is supposed to operate over tree
 1608 structured data. As it turns out, it is not possible to have fully homogeneous patches,
 1609 but we were able to identify homogeneous parts of our patches which we can use to
 1610 synchronize changes when defining our merge operation, but let us not get ahead of
 1611 ourselves.

1612 *Structural Patches* differ from edit-scripts by using tree-shaped, homogeneous patch
 1613 – a patch transforms two values of the same type. The edit operations themselves are
 1614 analogous to edit scripts, we support insertions, deletions and copies, but these are struc-
 1615 tured to follow the sums-of-products of datatypes: there is one way of changing sums,

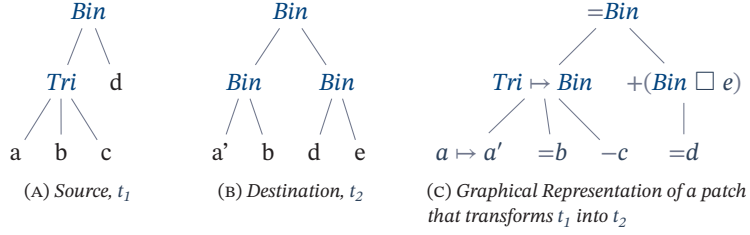


FIGURE 4.1: Graphical representation of a simple transformation. Copies, insertions and deletions around the tree are represented with $=$, $+$ and $-$ respectively. Modifications are denoted $\cdot \mapsto \cdot$.

one way of changing products and one way of changing the recursive positions of a value.
For example, consider the following trees:

$$\begin{aligned} t_1 &= \text{Bin} (\text{Tri } a \ b \ c) \ d \\ t_2 &= \text{Bin} (\text{Bin } a' \ b) \ (\text{Bin } d \ e) \end{aligned}$$

These are the trees that are depicted in Figure 4.1(A) and Figure 4.1(B) respectively. How should we represent the transformation mapping t_1 into t_2 ? Traversing the trees from their roots, we see that on the outermost level they both consist of a *Bin*, yet the fields of the source and destination nodes are different: the first field changes from a *Bin* to a *Tri*, which requires us to reconcile the list of fields $[a, b, c]$ into $[a', b]$. Which can be done by the means of an edit-script. The second field, however, witnesses a change in the recursive structure of the type. We see that we have inserted new information, namely $(\text{Bin } \square \ e)$. After inserting this *context*, we simply copy d from the source to the destination. This transformation has been sketched graphically in Figure 4.1(C), and showcases all the necessary pieces we will need to write a general encoding of transformations between objects that support insertions, deletions and copies.

The stdiff approach to differencing is unlike the edit-scripts we saw previously, using the shape of the datatype in question to define a structured notion of patch. As we will see in the remainder of this chapter, however, *computing* these patches is intractable. This lead us to abandon this approach in favor of the differencing algorithm presented in Chapter 5. Nonetheless, we believe there is value in studying this approach. For one it explores a different part in the design space compared to the *gdif* algorithm we saw previously, but it also provides insights that help understand the more efficient approach in Chapter 5.

To write the *stdiff* algorithms in Haskell, we must rely on the *generics-mrsop* library (Section 3.1) as our generic programming workhorse for two reasons. First, we do require the concept of explicit sums of products in the very definition of $\text{Patch}_{\text{ST}} \ x$.

Secondly, we need `gdiff`'s assistance in computing patches (Section 4.3.2) and `gdiff` also requires, to a lesser extent, sums of products structured datatypes, hence is easily written with `generics-mrsop`, as seen in Section 3.1.4.

The contributions in this chapter arise from joint work with Pierre-Evariste Dagand, published in TyDe 2017 [69] and coded in Agda [77]Agda repository¹. Later, we collaborated closely with a MSc student, Arian van Putten, in translating the Agda code to Haskell, extending its scope to mutually recursive datatypes. The code presented here, however, is loosely based on Van Putten's translation of our Agda repository to Haskell as part of his Master thesis work [86]. We chose to present all of our work in a single programming language to keep the thesis consistent throughout.

In this chapter we will delve into the construction of `PatchST` and its respective components. Firstly, we familiarize ourselves with `PatchST` and its application function, Section 4.1. Next we look into merging and its commutativity proof in Section 4.2. Lastly, we discuss the `diff` function in Section 4.3, which comprises a significant drawback of the `stdiff` approach for its computational complexity.

4.1 THE TYPE OF PATCHES

Next we look at the `PatchST` type, starting with a single layer of datatype, i.e., a single application of the datatypes pattern functor. Later, in Section 4.1.2 we extend this treatment to recursive datatypes, essentially by taking the fixpoint of the constructions in Section 4.1.1. The `generics-mrsop` library (Chapter 3) will be used throughout the exposition.

Recall that a datatype, when seen through its initial algebra semantics [98], can be seen as an infinite succession of applications of its pattern functor, call it F , to itself: $\mu F = F(\mu F)$. The `PatchST` type will describe the differences between values of μF by successively applying the description of differences between values of type F , closely following the initial algebra semantics of datatypes.

4.1.1 FUNCTORIAL PATCHES

Handling *one layer* of recursion is done by addressing the possible changes at the sum level, followed by some reconciliation at the product level when needed.

The first part of our algorithm handles the *sums* of the universe. Given two values, x and y , it computes the *spine*, capturing the largest common coproduct structure. We distinguish three possible cases:

¹<https://github.com/VictorCMiraldo/stdiff>

- 1673 • x and y are fully equal, in which case we copy the full values regardless of their
1674 contents. They must also be of the same type.
- 1675 • x and y have the same constructor – i.e., $x = \text{inj } c \text{ } px$ and $y = \text{inj } c \text{ } py$ – but some
1676 subtrees of x and y are distinct, in which case we copy the head constructor and
1677 handle all arguments pairwise.
- 1678 • x and y have distinct constructors, in which case we record a change in constructor
1679 and a choice of the alignment of the source and destination’s constructor fields.
1680 Here, x and y might be of a different type in the family.

1681 The datatype *Spine*, defined below, formalizes this description. The three cases we
1682 describe above correspond to the three constructors of *Spine*. When two values are not
1683 equal, we need to represent the differences somehow. If the values have the same con-
1684 structor we need to reconcile the fields of that constructor whereas if the values have
1685 different constructors we need to reconcile the products that make the fields of the con-
1686 structors. We index the datatype *Spine* by the sum codes it operates over because we
1687 need to lookup the fields of the constructors that have changed, and *align* them in the
1688 case of *SChg*. Alignments will be introduced shortly, for the time being, let us continue
1689 to focus on spines. Intuitively, spines act on sums and capture the “largest shared co-
1690 product structure”. Recall $\kappa :: \text{kon} \rightarrow *$ interprets the opaque types in the mutually
1691 recursive family in question and $\text{codes} :: [[\text{Atom kon}]]$ lists all the sums-of-products
1692 in the family, both come from `generics-mrsop` representation of mutually recursive
1693 datatypes, discussed in Section 3.1.

```

1694 data Spine  $\kappa$  codes :: [[Atom kon]]  $\rightarrow$  [[Atom kon]]  $\rightarrow$  * where
    Scp  :: Spine  $\kappa$  codes  $s_1$   $s_1$ 
    SCns :: Constr  $s_1$   $c_1$   $\rightarrow$  NP (At  $\kappa$  codes) (Lkup  $c_1$   $s_1$ )  $\rightarrow$  Spine  $\kappa$  codes  $s_1$   $s_1$ 
    SChg :: Constr  $s_1$   $c_1$   $\rightarrow$  Constr  $s_2$   $c_2$   $\rightarrow$  Al  $\kappa$  codes (Lkup  $c_1$   $s_1$ ) (Lkup  $c_2$   $s_2$ )
            $\rightarrow$  Spine  $\kappa$  codes  $s_1$   $s_2$ 

```

1695 Our Agda model [69] handles only regular types, or, mutually recursive families
1696 consisting of a single datatype. Hence, the *Spine* type would arise naturally as a homo-
1697 geneous type. While extending the Agda model to a full fledged Haskell implementation,
1698 together with Van Putten [86], we noted how this would severely limit the number of po-
1699 tential copy opportunities throughout patches. For example, imagine we want to patch
1700 the following values:

```

1701 data T = T1 X Y Z | T2 U
data U = U1 X Y Z | U2 T
diff (T1 x1 y1 z1) (U1 x2 y2 z2) = SChg T1 U1 ...

```

1702 With a fully homogeneous *Spine* type, our only option is to delete T_1 , then insert
1703 U_1 at the *recursion* layer (4.1.2) This would be unsatisfactory as it only allows copying

1704 of one of the fields, where `gdiff` would be able to copy more fields for it does not care
1705 about the recursive structure.

1706 The semantics of *Spine* are straightforward, but before continuing with *applySpine*,
1707 a short technical interlude is necessary. The *testEquality*, below, is used to compare the
1708 type indices for propositional equality. It comes from *Data.Type.Equality* and has type
1709 $f\ a \rightarrow f\ b \rightarrow \text{Maybe}\ (a \sim\!:\! b)$. Also note that we must pass two *SNat* arguments
1710 to disambiguate the *ix* and *iy* type variables. Without those arguments, these variables
1711 would only appear as an argument to a type family, which may not be injective and
1712 would trigger a type error. Using the *SNat* singleton [26] is the standard Haskell type-
1713 level programming workaround to this problem.

1714 `data SNat :: Nat → * where ...`

1715 The *applySpine* function is given by checking the provided value is made up with
1716 the required constructor. In the *SCns* case we must ensure that type indices match –
1717 for Haskell type families may not be injective – then simply map over the fields with the
1718 *applyAt* function, which applies changes to atoms. Otherwise, we reconcile the fields
1719 with the *applyAl* function, whose definition follow shortly.

```

1720
1721   applySpine :: (EqHO κ) ⇒ SNat ix → SNat iy
1722               → Spine κ codes (Lkup ix codes) (Lkup iy codes)
1723               → Rep κ (Fix κ codes) (Lkup ix codes)
1724               → Maybe (Rep κ (Fix κ codes) (Lkup iy codes))
1725   applySpine _ _ Scp x = return x
1726   applySpine ix iy (SCns c1 dxs) (sop → Tag c2 xs) = do
1727     Refl ← testEquality ix iy
1728     Refl ← testEquality c1 c2
1729     inj c2 <$> (mapNPM applyAt (zipNP dxs xs))
1730   applySpine _ _ (SChg c1 c2 al) (sop → Tag c3 xs) = do
1731     Refl ← testEquality' c1 c3
1732     inj c2 <$> applyAl al xs

```

1721 The *Spine* datatype and *applySpine* are responsible for matching the *constructors* of
1722 two trees, but we still need to determine how to continue representing the difference in
1723 the products of data stored therein. At this stage in our construction, we are given two
1724 heterogeneous lists, corresponding to the fields associated with two distinct construc-
1725 tors. As a result, these lists need not have the same length nor store values of the same
1726 type. To do so, we need to decide how to line up the constructor fields of the source and
1727 destination. We shall refer to the process of reconciling the lists of constructor fields as
1728 solving an *alignment* problem.

1729 Finding a suitable alignment between two lists of constructor fields amounts to find-
1730 ing a suitable *edit-script*, that relates source fields to destination fields. The *Al* datatype

below describes such edit-scripts for a heterogeneously typed list of atoms. These scripts may insert fields in the destination (*Ains*), delete fields from the source (*Adel*), or associate two fields from both lists (*AX*).

```

data Al κ codes :: [Atom kon] → [Atom kon] → * where
  A0  :: Al κ codes '[]' []
  AX  :: At κ codes x      → Al κ codes xs ys → Al κ codes (x ': xs) (x ': ys)
  Adel :: NA κ (Fix κ codes) x → Al κ codes xs ys → Al κ codes (x ': xs)      ys
  Ains :: NA κ (Fix κ codes) x → Al κ codes xs ys → Al κ codes      xs (x ': ys)

```

We require alignments to preserve the order of the arguments of each constructor, thus forbidding permutations of arguments. In effect, the datatype of alignments can be viewed as an intentional representation of (partial) *order and type preserving maps*, along the lines of McBride’s order preserving embeddings [60], mapping source fields to destination fields. This makes sure that our patches also give rise to tree mappings (Section 2.1.2) in the classical tree-edit distance sense.

Provided a partial embedding for atoms, we can therefore interpret alignments into a function transporting the source fields over to the corresponding destination fields, failure potentially occurring when trying to associate incompatible atoms. Recall (\times) and ϵ are the constructors of type *NP*:

```

applyAl :: (EqHO κ) ⇒ Al κ codes xs ys → PoA κ (Fix κ codes) xs
        → Maybe (PoA κ (Fix κ codes) ys)
applyAl A0      ε      = return ε
applyAl (AX dx dxs) (x × xs) = (×) <$> applyAt (dx ;*: x) <*> applyAl dxs xs
applyAl (Ains x dxs) xs = (x×) <$> applyAl dxs xs
applyAl (Adel x dxs) (y × xs) = guard (eq1 x y) *> applyAl dxs xs

```

Finally, when synchronizing atoms we must distinguish between a recursive position or opaque data. In case of opaque data, we simply record the old value and the new value.

```

data TrivialK (κ :: kon → *) :: kon → * where
  Trivial :: κ kon → κ kon → TrivialK κ kon

```

In case we are at a recursive position, we record a potential change in the recursive position with *Alμ*, which we will get to shortly.

```

data At (κ :: kon → *) (codes :: [[[Atom kon]]]) :: Atom kon → * where
  AtSet  :: TrivialK κ kon → At κ codes ('K kon)
  AtFix  :: (IsNat ix) ⇒ Alμ κ codes ix ix → At κ codes ('I ix)

```

1753 The application function for atoms follows the same structure. In case we are apply-
 1754 ing a patch to an opaque type, we must understand whether said patch represents a copy,
 1755 i.e., the source and destination values are the same. If that is the case, we simply copy
 1756 the provided value. Otherwise, we must ensure the provided value matches the source
 1757 value. The recursive position case is directly handled by the *applyAlμ* function.

$$\begin{aligned}
 & \text{applyAt} :: (\text{EqHO } ki) \Rightarrow \text{At } \kappa \text{ codes at} \rightarrow \text{NA } \kappa (\text{Fix } \kappa \text{ codes}) \text{ at} \\
 & \quad \rightarrow \text{Maybe } (\text{NA } \kappa (\text{Fix } \kappa \text{ codes}) \text{ at}) \\
 & \text{applyAt } (\text{AtSet } (\text{Trivial } x \ y)) (\text{NA}_K \ a) \\
 & \quad | \text{eqHO } x \ y = \text{Just } (\text{NA}_K \ a) \\
 & \quad | \text{eqHO } x \ a = \text{Just } (\text{NA}_K \ b) \\
 & \quad | \text{otherwise} = \text{Nothing} \\
 & \text{applyAt } (\text{AtFix } px) (\text{NA}_I \ x) = \text{NA}_I \ \langle \$ \rangle \ \text{applyAl}\mu \ px \ x
 \end{aligned}$$

1759 The last step is to address how to make changes over the recursive structure of our
 1760 value, defining *Alμ* and *applyAlμ*, which will be our next concern.

1761

1762 4.1.2 RECURSIVE CHANGES

1763 In the previous section, we presented patches describing changes to the coproducts, prod-
 1764 ucts, and atoms of our *SoP* universe. This treatment handled just a single layer of the fix-
 1765 point construction. In this section, we tie the knot and define patches describing changes
 1766 to arbitrary *recursive* datatypes.

1767 To represent generic patches on values of *Fix codes ix*, we will define two mutually
 1768 recursive datatypes *Alμ* and *Ctx*. The semantics of both these datatypes will be given by
 1769 defining how to *apply* them to arbitrary values:

- 1770 • Much like alignments for products, a similar phenomenon appears at fixpoints.
 1771 When comparing two recursive structures, we can insert, remove or modify con-
 1772 structors. Since we are working over mutually recursive families, removing or
 1773 inserting constructors can change the overall type. We will use *Alμ ix iy* to spec-
 1774 ify these edit-scripts at the constructor-level, describing a transformation from
 1775 *Fix codes ix* to *Fix codes iy*.
- 1776 • Whenever we choose to insert or delete a recursive subtree, we must specify *where*
 1777 this modification takes place. To do so, we will define a new type *Ctx ... ::*
 1778 *'[Atom kon] → **, inspired by zippers [41, 59], to navigate through our data-
 1779 structures. A value of type *Ctx ... p* selects a single atom *I* from the product of
 1780 type *p*.

Modeling changes over fixpoints closely follows our definition of alignments of products. Instead of inserting and deleting elements of the product we insert, delete or modify *constructors*. Our previous definition of spines merely matched the constructors of the source and destination values – but never introduced or removed them. It is precisely these operations that we must account for here.

```

1781 data  $Al\mu \kappa \text{ codes} :: Nat \rightarrow Nat \rightarrow *$  where
1782    $Spn :: Spine \kappa \text{ codes} (Lkup \text{ ix codes}) (Lkup \text{ iy codes})$ 
1783      $\rightarrow Al\mu \kappa \text{ codes ix iy}$ 
1784    $Ins :: Constr (Lkup \text{ iy codes}) c \rightarrow InsCtx \kappa \text{ codes ix} (Lkup c (Lkup \text{ iy codes}))$ 
1785      $\rightarrow Al\mu \kappa \text{ codes ix iy}$ 
1786    $Del :: Constr (Lkup \text{ ix codes}) c \rightarrow DelCtx \kappa \text{ codes iy} (Lkup c (Lkup \text{ ix codes}))$ 
1787      $\rightarrow Al\mu \kappa \text{ codes ix iy}$ 

```

The first constructor, *Spn*, does not perform any new insertions and deletions, but instead records a spine and an alignment of the underlying product structure. This closely follows the patches we have seen in the previous section. To insert a new constructor, *Ins*, requires two pieces of information: a choice of the new constructor to be introduced, called *c*, and the fields associated with that constructor. Note that we only need to record *all but one* of the constructor's fields, as represented by a value of type $InsCtx \text{ ki codes ix} (Lkup c (Lkup \text{ iy codes}))$. Deleting a constructor is analogous to insertions, with *InsCtx* and *DelCtx* being slight variations over *Ctx*, where one actually flips the arguments to ensure the transformation is on the right direction.

```

1797 type  $InsCtx \kappa \text{ codes} = Ctx \kappa \text{ codes} \quad (Al\mu \kappa \text{ codes})$ 
1798 type  $DelCtx \kappa \text{ codes} = Ctx \kappa \text{ codes} (Flip (Al\mu \kappa \text{ codes}))$ 
1799 newtype  $Flip f \text{ ix iy} = Flip \{unFlip :: f \text{ iy ix}\}$ 

```

Our definition of insertion and deletions relies on identifying *one* recursive argument among the product of possibilities. To model this accurately, we define an indexed zipper to identify a recursive atom (indicated by a value of type *I*) within a product of atoms. Conversely, upon deleting a constructor from the source structure, we exploit *Ctx* to indicate find the subtree that should be used for the remainder of the patch application, discarding all other constructor fields. We parameterize the *Ctx* type with a $Nat \rightarrow Nat \rightarrow *$ argument to distinguish between these two cases, as seen above.

```

1804 data  $Ctx \kappa \text{ codes} (p :: Nat \rightarrow Nat \rightarrow *) (ix :: Nat) :: [Atom \text{ kon}] \rightarrow *$  where
1805    $H :: (IsNat \text{ iy}) \Rightarrow p \text{ ix iy} \rightarrow PoA \kappa (Fix \kappa \text{ codes}) \text{ xs} \rightarrow Ctx \kappa \text{ codes } p \text{ ix} ('I \text{ iy} ' : \text{ xs})$ 
1806    $T :: NA \kappa (Fix \kappa \text{ codes}) a \rightarrow Ctx \kappa \text{ codes } p \text{ ix xs} \rightarrow Ctx \kappa \text{ codes } p \text{ ix} (a ' : \text{ xs})$ 

```

Consequently, we will have two application functions for contexts, one that inserts and one that removes contexts. This makes clear the need to flip the type indexes of *Alμ*

1807 when defining *DelCtx*. Inserting a context is done by receiving a tree and returning the
 1808 product stored in the context with the distinguished field applied to the received tree:

1809
$$\begin{aligned} \text{insCtx} &:: (\text{IsNat } ix, \text{EqHO } \kappa) \Rightarrow \text{InsCtx } \kappa \text{ codes } ix \text{ xs} \rightarrow \text{Fix } \kappa \text{ codes } ix \\ &\rightarrow \text{Maybe } (\text{PoA } \kappa (\text{Fix } \kappa \text{ codes}) \text{ xs}) \\ \text{insCtx } (H \ x \ \text{rest}) \ v &= (\times \text{rest}) \circ \text{NA}_I \langle \$ \rangle \text{ applyAl}\mu \ x \ v \\ \text{insCtx } (T \ a \ \text{ctx}) \ v &= (a \times) \langle \$ \rangle \text{ insCtx } \text{ctx} \ v \end{aligned}$$

1810 The deletion function discards any information we have about all the constructor
 1811 fields, except for the subtree used to continue the patch application process. This is a
 1812 consequence of our design decision, at the time, of having application functions as per-
 1813 missive as possible. Intuitively, the deletion context identifies the only field that should
 1814 not be deleted. By not checking whether the elements we are applying to match the ones
 1815 that should be deleted, we get an application function that applies to more elements for
 1816 free.

1817
$$\begin{aligned} \text{delCtx} &:: (\text{IsNat } ix, \text{EqHO } \kappa) \Rightarrow \text{DelCtx } \kappa \text{ codes } ix \text{ xs} \rightarrow \text{PoA } \kappa (\text{Fix } \kappa \text{ codes}) \text{ xs} \\ &\rightarrow \text{Maybe } (\text{Fix } \kappa \text{ codes } ix) \\ \text{delCtx } (H \ x \ \text{rest}) \ (\text{NA}_I \ v \times p) &= \text{applyAl}\mu \ (\text{unFlip } x) \ v \\ \text{delCtx } (T \ a \ \text{ctx}) \ (\text{at } \quad \times p) &= \text{delCtx } \text{ctx} \ p \end{aligned}$$

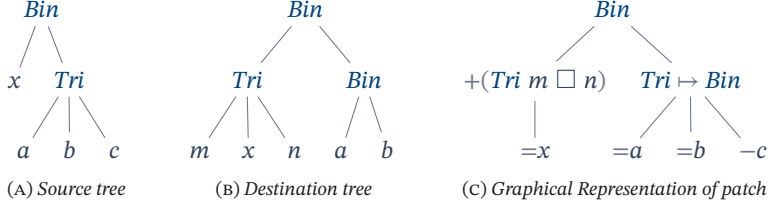
1818 Finally, the application function for *Al* μ is nothing but selecting whether we should
 1819 use the spine functionality or insertion and deletion of a context.

1820
$$\begin{aligned} \text{applyAl}\mu &:: (\text{IsNat } ix, \text{IsNat } iy, \text{EqHO } \kappa) \Rightarrow \text{Al}\mu \ \kappa \text{ codes } ix \text{ iy} \rightarrow \text{Fix } \kappa \text{ codes } ix \\ &\rightarrow \text{Maybe } (\text{Fix } \kappa \text{ codes } iy) \\ \text{applyAl}\mu \ (\text{Spn } sp) \ (\text{Fix } rep) &= \text{Fix } \langle \$ \rangle \text{ applySpine } _ _ \text{ spine } rep \\ \text{applyAl}\mu \ (\text{Ins } c \ \text{ctx}) \ (\text{Fix } rep) &= \text{Fix } \circ \text{inj } c \langle \$ \rangle \text{ insCtx } \text{ctx} \ f \\ \text{applyAl}\mu \ (\text{Del } c \ \text{ctx}) \ (\text{Fix } rep) &= \text{delCtx } \text{ctx} \langle \$ \rangle \text{ match } c \ rep \end{aligned}$$

1821 The two underscores at the *Spn* case are just an extraction of the necessary singletons
 1822 to make the *applySpine* typecheck. These can be easily replaced by *getSNat* with the
 1823 correct proxies. Figure 4.2 provides a graphical illustration of a value of type *Patch*_{ST}
 1824 that transforms two concrete trees.

1825
$$\begin{aligned} \text{type Patch}_{\text{ST}} \ \kappa \text{ codes } ix &= \text{Al}\mu \ \kappa \text{ codes } ix \text{ ix} \\ \text{apply}_{\text{ST}} &:: (\text{IsNat } ix, \text{EqHO } \kappa) \Rightarrow \text{Patch}_{\text{ST}} \ \kappa \text{ codes } ix \rightarrow \text{Fix } \text{codes } ix \rightarrow \text{Maybe } (\text{Fix } \text{codes } ix) \\ \text{apply}_{\text{ST}} &= \text{applyAl}\mu \end{aligned}$$

1826 An easily overlooked property of our patch definition is that the destination values it
 1827 computes are guaranteed to be type-correct *by construction*. This is unlike the line-based
 1828 or untyped approaches (which may generate ill-formed values) and similar to earlier
 1829 results on type-safe differences [51].



```

Spn (SCns Bin ( AtFix (Ins Tri (T m (H (Spn Cpy) (n × ε))))
      × AtFix (Spn (Schg Tri Bin (AX (AtFix (Spn Cpy)) (
      AX (AtFix (Spn Cpy)) (
      ADel (NAI c)
      A0))))))
ε))

```

(D) $Patch_{ST}$ that transforms source into destination, in Haskell

FIGURE 4.2: A value of type $Patch_{ST}$ with its graphical representation.

4.2 MERGING PATCHES

The patches encoded in the $Patch_{ST}$ type clearly identify a prefix of constructors copied from the root of a tree up until the location of the changes and any insertion or deletions that might happen along the way. Moreover, since these patches also mirror the tree structure of the data in question, it becomes quite natural to identify separate changes. For example, if one change works on the left subtree of the root, and another on the right, they are clearly disjoint and can be merged. Finally, the explicit representation of insertions and deletions at the fixpoint level gives us a simple global alignment for our synchronizer.

In this section we discuss a simple merging algorithm, which reconciles changes from two different patches whenever these are *non-interfering*, for example, as in Figure 4.3. We call non-interfering patches *disjoint*, as they operate on separate parts of a tree.

A positive aspect of the $Patch_{ST}$ approach in comparison with a purely edit-scripts based approach is the significantly simpler merge function. This is due to $Patch_{ST}$ being having clear homogeneous sections. Consequently, the type of the merge function is simple and reflects the fact that we expect a patch that operates over the values of the same type as a result:

```

merge :: PatchST κ codes ix → PatchST κ codes ix → Maybe (PatchST κ codes ix)

```

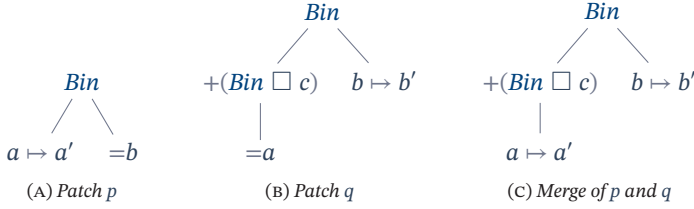


FIGURE 4.3: A simple example of mergeable patches.

1849 A call to *merge*, in Haskell, returns *Nothing* if the patches have non-disjoint changes,
 1850 that is, if both patches want to change the *same part* of the source tree.

1851 Prior to prototyping *stdiff* in Haskell, we already had a working model of *stdiff*
 1852 in Agda [69], which was created with the goal of proving that the merging algorithm
 1853 would respect locality. In our Agda model, we have divided the merge function and
 1854 the notion of disjointness, which yields a total merge function for the subset of disjoint
 1855 patches:

1856 $\text{merge} : (p\ q : \text{Patch } \kappa\ \text{codes } ix) \rightarrow \text{Disjoint } p\ q \rightarrow \text{Patch } \kappa\ \text{codes } ix$

1857 A value of type *Disjoint* $p\ q$ corresponds to a proof that p and q change different
 1858 parts of the source tree and is a symmetric relation – that is, *Disjoint* $p\ q$ iff *Disjoint* $q\ p$.
 1859 This separation makes reasoning about the merge function much easier. In fact, we
 1860 have proven that the merge function over regular datatypes commutes. A simplified
 1861 statement of our theorem is given below:

1862 $\begin{aligned} \text{merge-commutes} & : (p\ q : \text{Patch } \kappa\ \text{codes } ix) \\ & \rightarrow (\text{hyp} : \text{Disjoint } p\ q) \\ & \rightarrow \text{apply } (\text{merge } p\ q\ \text{hyp}) \circ q \equiv \text{apply } (\text{merge } q\ p\ (\text{sym } \text{hyp})) \circ p \end{aligned}$

1863 It is also worth noting that encoding the *merge* to be applied to the divergent repli-
 1864 cas instead of the common ancestor – *residual-like* approach to merging, Section 2.1.4 –
 1865 is instrumental to write a concise property and, consequently, prove the result. A merge
 1866 function that applies to the common ancestor would probably require a much more con-
 1867 voluted encoding of *merge-commutes* above.

1868 In a Haskell development, however, it is simpler to rely on the *Maybe* monad for
 1869 disjointness. In fact, we define disjointness as whether or not merge returns a *Just*:

1870 $\begin{aligned} \text{disjoint} & :: \text{Patch } \kappa\ \text{codes } ix \rightarrow \text{Patch } \kappa\ \text{codes } ix \rightarrow \text{Bool} \\ \text{disjoint } p\ q & = \text{maybe } (\text{const } \text{True})\ \text{False } (\text{merge } p\ q) \end{aligned}$

1871 The definition of the *merge* function is given in its entirety in Figure 4.4, but we
 1872 discuss some interesting cases inline next. For example, when one change deletes a con-
 1873 structor but the other performs a change within said constructor we must check that they
 1874 operate over *the same* constructor. When that is the case, we must go ahead and ensure
 1875 the deletion context, *ctx*, and the changes in the product of atoms, *at*, are compatible.

1876 $\text{merge } (\text{Del } c_1 \text{ ctx}) (\text{Spn } (\text{SCns } c_2 \text{ at})) = \text{testEquality } c_1 \text{ } c_2 \succ \lambda \text{Refl} \rightarrow \text{mergeCtxAt } \text{ctx} \text{ } \text{at}$

1877 A (deletion) context is disjoint from a list of atoms if the patch in the hole of the con-
 1878 text returns the same type of element than the patch on the product of patches and they
 1879 are both disjoint. Moreover, the rest of the product of patches must consist in identity
 1880 patches. Otherwise, we risk deleting newly introduced information.

1881 $\begin{aligned} &\text{mergeCtxAt} :: \text{DelCtx } \kappa \text{ codes } \text{iy } \text{xs} \rightarrow \text{NP } (\text{At } \kappa \text{ codes}) \text{ xs} \rightarrow \text{Maybe } (\text{Almu } \kappa \text{ codes } \text{ix } \text{iy}) \\ &\text{mergeCtxAt } (\text{H } (\text{AlmuMin } \text{almu}' \text{ rest}) (\text{AtFix } \text{almu} \times \text{xs})) = \text{do} \\ &\quad \text{Refl} \leftarrow \text{testEquality } (\text{almuDest } \text{almu}) (\text{almuDest } \text{almu}') \\ &\quad x \leftarrow \text{mergeAlmu } \text{almu}' \text{ } \text{almu} \\ &\quad \text{guard } (\text{and } \$ \text{elimNP identityAt } \text{xs}) \\ &\quad \text{pure } x \\ &\text{mergeCtxAt } (\text{T } \text{at } \text{ctx}) (x \times \text{xs}) = \text{guard } (\text{identityAt } x) \succ \text{mergeCtxAt } \text{ctx} \text{ } \text{xs} \end{aligned}$

1882 The *testEquality* is there to ensure the patches to be merged are producing the same
 1883 element of the mutually recursive family. This is one of the two places where we need
 1884 these checks when adapting our Agda model to work over mutually recursive types. The
 1885 second adaptation is shown shortly.

1886 The *mergeAtCtx* function, dual to *mergeCtxAt*, merges a *NP (At κ codes) xs* and
 1887 a *DelCtx κ codes iy xs* into a *Maybe (DelCtx κ codes iy xs)*, essentially preserving
 1888 the *T at* it finds on the recursive calls. Another interesting case happens on one of the
 1889 *mergeSpine* cases, whose full implementation can be seen in Figure 4.5. The *SChg* over
 1890 *SCns* case must ensure we are working over the same element of the mutually recursive
 1891 family, with a *testEquality ix iy*. This is the second place where we need to adapt the
 1892 code in the Agda repository to work over mutually recursive types.

1893 $\begin{aligned} &\text{mergeSpine} :: \text{SNat } \text{ix} \rightarrow \text{SNat } \text{iy} \\ &\quad \rightarrow \text{Spine } \kappa \text{ codes } (\text{Lkup } \text{ix codes}) (\text{Lkup } \text{iy codes}) \\ &\quad \rightarrow \text{Spine } \kappa \text{ codes } (\text{Lkup } \text{ix codes}) (\text{Lkup } \text{iy codes}) \\ &\quad \rightarrow \text{Maybe } (\text{Spine } \kappa \text{ codes } (\text{Lkup } \text{ix codes}) (\text{Lkup } \text{iy codes})) \\ &\text{mergeSpine } \text{ix } \text{iy} (\text{SChg } \text{cx } \text{cy } \text{al}) (\text{SCns } \text{cz } \text{zs}) = \text{do } \text{Refl} \leftarrow \text{testEquality } \text{ix } \text{iy} \\ &\quad \text{Refl} \leftarrow \text{testEquality } \text{cx } \text{cz} \\ &\quad \text{SCns } \text{cy} \prec \$ \text{mergeAlAt } \text{al } \text{zs} \end{aligned}$


```

-- Non-disjoint recursive spines:
merge (Ins _ _) (Ins _ _) = Nothing
merge (Spn (SCHg _ _)) (Del _ _) = Nothing
merge (Del _ _) (Spn (Schg _ _)) = Nothing
merge (Del _ _) (Del _ _) = Nothing

-- Obviously disjoint recursive spines:
merge (Spn Scp) (Del c2 s2) = Just (Del c2 s2)
merge (Del c1 s2) (Spn Scp) = Just (Spn Scp)

-- Spines might be disjoint from spines and deletions:
merge (Spn s1) (Spn s2)
  = Spn <$> mergeSpine (getSNat (Proxy@ix)) (getSNat (Proxy@iy)) s1 s2
merge (Spn (SCns c1 at1)) (Del c2 s2)
  = Del c1 <$> mergeAtCtx at1 s2
merge (Del c1 s1) (Spn (SCns c2 at2))
  = do Refl ← testEquality c1 c2 -- disjoint if same constructor
    mergeCtxAt s1 at2

-- Insertions are disjoint from anything except insertions.
-- Overall disjointness does depend on the recursive parts, though.
merge (Ins c1 s1) (Spn s2) = Spn ∘ SCns c1 <$> mergeCtxAlmu s1 (Spn s2)
merge (Ins c1 s1) (Del c2 s2) = Spn ∘ SCns c1 <$> mergeCtxAlmu s1 (Del c2 s2)
merge (Spn s1) (Ins c2 s2) = Ins c2 <$> (mergeAlmuCtx (Spn s1) s2)
merge (Del c1 s1) (Ins c2 s2) = Ins c2 <$> (mergeAlmuCtx (Del c1 s1) s2)

```

FIGURE 4.4: Definition of merge

```

-- Non-disjoint spines:
mergeSpine _ _ (SCHg _ _ _) (SCHg _ _ _) = Nothing

-- Obviously disjoint spines:
mergeSpine _ _ Scp s = Just s
mergeSpine _ _ s Scp = Just Scp

-- Disjointness depends on recursive parts:
mergeSpine _ _ (SCns cx xs) (SCns cy ys) = do Refl ← testEquality cx cy
  SCns cx <$> mergeAts xs ys
mergeSpine _ _ (SCns cx xs) (SCHg cy cz al) = do Refl ← testEquality cx cy
  SCHg cy cz <$> mergeAtAl xs al
mergeSpine ix iy (SCHg cx cy al) (SCns cz zs) = do Refl ← testEquality ix iy
  Refl ← testEquality cx cz
  SCns cy <$> mergeALAt al zs

```

FIGURE 4.5: Definition of mergeSpine

1894 4.3 COMPUTING $Patch_{ST}$

1895 In the previous sections, we have devised a typed representation for differences. We have
 1896 seen that this representation is interesting in and by itself: being richly-structured and
 1897 typed, it can be thought of as a non-trivial programming language whose denotation is
 1898 given by the application function. Moreover, we have seen how to merge two disjoint
 1899 differences. However, as programmers, we are mainly interested in *computing* patches
 1900 from a source and a destination. Unfortunately, however, this is where the good news
 1901 stops. Computing a value of type $Patch_{ST}$ is computationally expensive and represents
 1902 one of the main downsides of this approach.

1903 In this section we explore our attempts at computing differences with the `stdiff`
 1904 framework. We start by outlining a nondeterministic specification of an algorithm for
 1905 computing a $Patch_{ST}$, in Section 4.3.1. We then provide example algorithms that imple-
 1906 mented said specification in Section 4.3.2. All these approaches however, we will always
 1907 need to make choices. Moreover, the rich structure of $Patch_{ST}$ makes a memoized algo-
 1908 rithm much more difficult to be written. Consequently, computing a $Patch_{ST}$ will always
 1909 be a computationally inefficient process, rendering it unusable in practice.

1910 4.3.1 NAIVE ENUMERATION

1911 The simplest option for computing a patch that transforms a tree x into y is enumerating
 1912 all possible patches and filtering out those with the smallest *cost*, for some *cost* metric.
 1913 In this section, we will write a naive enumeration engine for $Patch_{ST}$ and argue that
 1914 regardless of the *cost* notion, the state space explodes quickly and becomes intractable.

1915 The enumeration follows the Agda model [69] closely and is not very surprising.
 1916 Nevertheless, it does act as a good specification for a better implementation later. Just
 1917 like for the linear case, the changes that can transform two values x and y of a given
 1918 mutually recursive family into one another are the deletion of a constructor from x , the
 1919 insertion of a constructor from y or changing the constructor of x into the one from y , as
 1920 witnessed by the `enumAl μ` function below.

$$\begin{aligned} enumAl\mu &:: Fix\ ki\ codes\ ix \rightarrow Fix\ ki\ codes\ iy \rightarrow [Al\mu\ ki\ codes\ ix\ iy] \\ enumAl\mu\ x\ y &= enumDel\ (sop\ \$\ unFix\ x)\ y \\ &\quad <|>\ enumIns\ x\ (sop\ \$\ unFix\ y) \\ &\quad <|>\ Spn\ <\$>\ enumSpn\ (snatFixIdx\ x)\ (snatFixIdx\ y) \\ &\quad (unFix\ x)\ (unFix\ y) \end{aligned}$$

where

$$\begin{aligned} enumDel\ (Tag\ c\ p)\ y_0 &= Del\ c\ <\$>\ enumDelCtx\ p\ y_0 \\ enumIns\ x_0\ (Tag\ c\ p) &= Ins\ c\ <\$>\ enumInsCtx\ x_0\ p \end{aligned}$$

1922 Enumerating all the patches from a deletion context of a given product p against
 1923 some fixpoint y consists of enumerating the patches that transform all of the fields of p
 1924 into y . The handling of insertion contexts is analogous, hence it is omitted here. Recall
 1925 that the *AlmuMin*, below, is used to flag the resulting context as a deletion context.

```

enumDelCtx :: PoA ki (Fix ki codes) prod → Fix ki codes iy → [DelCtx ki codes iy prod]
enumDelCtx Nil           = []
enumDelCtx (NAK x × xs) f = T (NAK x) <$> enumDelCtx xs f
enumDelCtx (NAI x × xs) f = (flip H xs ∘ AlmuMin) <$> enumAlμ x f
                        <|> T (NAI x) <$> enumDelCtx xs f
  
```

1927 Next we look into enumerating the spines between x and y , that is, changes to the
 1928 coproduct structure from x to y . Unlike our Agda model, we need to know over which
 1929 element of the mutually recursive family we are operating. This will dictate which con-
 1930 structors from *Spine* we are allowed to use. We gather this information through two
 1931 auxiliary *SNat* parameters. The choice of which spine constructor to use is determinis-
 1932 tic, that is, each case is uniquely determined by a *Spine* constructor.

```

enumSpn :: SNat ix → SNat iy
          → Rep ki (Fix ki codes) (Lkup ix codes)
          → Rep ki (Fix ki codes) (Lkup iy codes)
          → [Spine ki codes (Lkup ix codes) (Lkup iy codes)]
enumSpn six siy x y =
  let Tag cx px = sop x
      Tag cy py = sop y
  in case testEquality six siy of
    Nothing → SChg cx cy <$> enumAl px py
    Just Refl → case testEquality cx cy of
      Nothing → SChg cx cy <$> enumAl px py
      Just Refl → if eqHO px py
                    then return Scp
                    else SCns cx <$> mapNPM (uncurry' enumAt) (zipNP px py)
  
```

1934 Enumerating atoms, *enumAt*, is trivial. Atoms are either opaque types or recursive
 1935 positions. Opaque types are handled by *TrivialK* and recursive positions are handled
 1936 recursively by *enumAlμ*.

```

enumAt :: NA ki (Fix ki codes) at → NA ki (Fix ki codes) at → [At ki codes at]
enumAt (NAI x) (NAI y) = AtFix <$> enumAlμ x y
enumAt (NAK x) (NAK y) = return $ AtSet (Trivial x y)
  
```

1938 Finally, alignments of products is analogous to the longest common subsequence,
 1939 except that we must make sure that we only synchronize atoms with *AX* if they have the

1940 same type. The *enumAl* below illustrates the non-deterministic enumeration of align-
 1941 ments over two products-of-atoms.

```

enumAl :: PoA ki (Fix ki codes) p1 → PoA ki (Fix ki codes) p2 → [Al ki codes p1 p2]
enumAl Nil Nil = return A0
enumAl (x × xs) Nil = ADel x <$> enumAl xs Nil
enumAl Nil (y × ys) = AIns y <$> enumAl Nil ys
1942 enumAl (x × xs) (y × ys) = (ADel x <$> enumAl xs (y × ys))
                                <|> (AIns y <$> enumAl (x × xs) ys)
                                <|> case testEquality x y of
                                    Just Refl → AX <$> (enumAt x y) <*> enumAl xs ys
                                    Nothing → mzero

```

1943 From the definitinos of *enumAlμ* and *enumAl*, it is clear why this algorithm explodes
 1944 and becomes intractable. In *enumAlμ* we must choose between inserting, deleting or
 1945 copying a recursive constructor. In case we chose to copy a constructor, we then might
 1946 call *enumAl*, where we must chose between inserting, deleting or copying fields of con-
 1947 structors. We must enumerate these options for virtually each pair of constructors in the
 1948 source and destination trees.

1949 4.3.2 TRANSLATING FROM GDIFF

1950 Since enumerating all possible patches and then filtering a chosen one is time consuming
 1951 and requires an complex notion of cost over *Patch*_{ST}, it was clear we should be pursuing
 1952 better algorithms for our *diff* function. We have attempted two similar approaches to
 1953 filter the uninteresting patches out and optimize the search space.

1954 A first idea, which arose in conjuncton with Pierre-Evariste Dagand (private commu-
 1955 nication), was to use the already existing UNIX *diff* tool as some sort of *oracle*. That
 1956 is, we should only consider inserting and deleting elements that fall on lines marked as
 1957 such by UNIX *diff*. This idea was translated into Haskell by Garuffi [33], but the per-
 1958 formance was still very poor and computing the *Patch*_{ST} of two real-world Clojure files
 1959 still required several minutes.

1960 From Garuffi's experiments [33] we learnt that simply restricting the search space
 1961 was not sufficient. Besides the complexity introduced by arbitrary heuristics, using the
 1962 UNIX *diff* to flag elements of the AST was still too coarse. For one, the UNIX *diff*
 1963 can insert and delete the same line in some situations. Secondly, many elements of the
 1964 AST may fall on the same line.

1965 The second option is related, but instead of using a line-based oracle, we can use
 1966 *gdiff* Section 3.1.4 as the oracle, enabling us to annotate every node of the source and
 1967 destination trees with a information about whether that node was copied or not. This
 1968 strategy was translated into Haskell by Van Putten [86] as part of his MSc work. The gist

1969 of it is that we can use annotated fixpoints to tag each constructor of a tree with added
 1970 information. In this case, we are interested in whether this node would be copied or not
 1971 by `gdifff`:

1972 **data** *Ann* = *Modify* | *Copy*

1973 A *Modify* annotation corresponds to a deletion or insertion depending on whether
 1974 it is the source or destination tree respectively. Recall that an edit-script produced by
 1975 `gdifff` has type *ES* κ *codes* *xs* *ys*, where *xs* is the list of types of the source trees and *ys* is
 1976 the list of types of the destination trees. The definition of *ES* – introduced in Section 3.1.4
 1977 – is repeated below.

data *ES* κ *codes* :: [*Atom* *kon*] \rightarrow [*Atom* *kon*] \rightarrow * **where**
ES0 :: *ES* κ *codes* '[]' '[]'
 1978 *Ins* :: *Cof* κ *codes* *a* *t* \rightarrow *ES* κ *codes* i (*t* : $\#$: *j*) \rightarrow *ES* κ *codes* i (*a* ' : *j*)
Del :: *Cof* κ *codes* *a* *t* \rightarrow *ES* κ *codes* (*t* : $\#$: *i*) j \rightarrow *ES* κ *codes* (*a* ' : *i*) j
Cpy :: *Cof* κ *codes* *a* *t* \rightarrow *ES* κ *codes* (*t* : $\#$: *i*) (*t* : $\#$: *j*) \rightarrow *ES* κ *codes* (*a* ' : *i*) (*a* ' : *j*)

1979 Given a value of type *ES* κ *codes* *xs* *ys*, we have information about which construc-
 1980 tors of the trees in *NP* (*NA* κ (*Fix* κ *codes*)) *xs* should be copied. Our objective then is to
 1981 annotated the trees with this very information. This is done by the *annSrc* and *annDst*
 1982 functions. We will only look at *annSrc*, the definition of *annDst* is symmetric.

1983 Annotating the source forest with a given edit-script consists in matching which con-
 1984 structors present in the forest correspond to a copy and which correspond to a deletion.
 1985 The insertions in the edit-script concern the destination forest only. The *annSrc* func-
 1986 tion, below, does exactly that, proceeding by induction on the edit-script.

annSrc :: *NP* (*NA* κ (*Fix* κ *codes*)) *xs* \rightarrow *ES* κ *codes* *xs* *ys*
 \rightarrow *NP* (*NA* κ (*FixAnn* κ *codes* (*Const Ann*))) *xs*
annSrc *xs* *ES0* = *Nil*
annSrc *Nil* – = *Nil*
 1987 *annSrc* *xs* (*Ins* *c* *es*) = *annSrc'* *xs* *es*
annSrc (*x* \times *xs*) (*Del* *c* *es*) = **let** *poa* = *fromJust* \$ *matchCof* *c* *x*
in *insCofAnn* *c* (*Const Modify*) (*annSrc'* (*appendNP* *poa* *xs*) *es*)
annSrc' (*x* \times *xs*) (*Cpy* – *c* *es*) = **let** *poa* = *fromJust* \$ *matchCof* *c* *x*
in *insCofAnn* *c* (*Const Copy*) (*annSrc'* (*appendNP* *poa* *xs*) *es*)

1988 The deterministic diff function for *Al μ* starts by checking the annotations present at
 1989 the root of its argument trees. In case both are copies, we start with a spine. If at least
 1990 one of them is not a copy we insert or delete the constructor not flagged as a copy. We
 1991 must guard for the case that there exists a copy in the inserted or deleted subtree. In
 1992 case that does not hold, we would not be able to choose an argument of the inserted
 1993 or deleted constructor to continue diffing against, in *diffCtx*. When there are no more

1994 copies to be performed, we just return a *stiff* patch, which deletes the entire source and
 1995 inserts the entire destination tree.

```

diffAlmu :: FixAnn  $\kappa$  codes (Const Ann) ix  $\rightarrow$  FixAnn  $\kappa$  codes (Const Ann) iy
           $\rightarrow$  Al $\mu$   $\kappa$  codes ix iy
diffAlmu x@(FixAnn ann1 rep1) y@(FixAnn ann2 rep2) =
  case (getAnn ann1, getAnn ann2) of
    (Copy, Copy)    $\rightarrow$  Spn (diffSpine (getSNat $ Proxy@ix)
                                     (getSNat $ Proxy@iy)
                                     rep1 rep2)
    (Copy, Modify)  $\rightarrow$  if hasCopies y then diffIns x rep2
                                     else stiffAlmu (forgetAnn x) (forgetAnn y)
    (Modify, Copy)  $\rightarrow$  if hasCopies x then diffDel rep1 y
                                     else stiffAlmu (forgetAnn x) (forgetAnn y)
    (Modify, Modify)  $\rightarrow$  if hasCopies x then diffDel rep1 y
                                     else stiffAlmu (forgetAnn x) (forgetAnn y)
  where
    diffIns x rep = case sop rep of Tag c ys  $\rightarrow$  Ins c (diffCtx CtxIns x ys)
    diffDel rep y = case sop rep of Tag c xs  $\rightarrow$  Del c (diffCtx CtxDel y xs)

```

1997 The *diffCtx* function selects an element of a product to continue diffing against. We
 1998 naturally select the element that has the most constructors marked for copy as the ele-
 1999 ment we continue diffing against. The other fields of the product are placed on the *rigid*
 2000 part of the context, that is, the trees that will be deleted or inserted entirely, without
 2001 sharing any of their subtrees.

```

diffCtx :: InsOrDel  $\kappa$  codes p  $\rightarrow$  FixAnn  $\kappa$  codes (Const Ann) ix
           $\rightarrow$  NP (NA  $\kappa$  (FixAnn  $\kappa$  codes (Const Ann))) xs
           $\rightarrow$  Ctx  $\kappa$  codes p ix xs

```

2003 The other functions for translating two *FixAnn* κ codes (Const Ann) ix into a
 2004 $Patch_{ST}$ are straightforward and follow a similar reasoning process: extract the anno-
 2005 tations and defer copies until both source and destination annotation flag a copy.

2006 This version of the *diff* function runs in $\mathcal{O}(n^2)$ time, where n is the the number of
 2007 constructors in the bigger input tree. Although orders of magnitude better than naive
 2008 enumeration or using the UNIX `diff` as an oracle, a quadratic algorithm is still not prac-
 2009 tical, particularly when n tens do be large – real-world source files have tens of thousands
 2010 abstract syntax elements.

2011 4.4 DISCUSSION

2012 With `stdiff` we learned that the difficulties of edit-script based approaches are not due,
2013 exclusively, to using linear data to represent transformations to tree structured data. An-
2014 other important aspect that we unknowingly overlooked, and ultimately did lead to a
2015 prohibitively expensive *diff* function, was the necessity to choose a single copy opportu-
2016 nity. This happens whenever a subtree could be copied in two or more different ways,
2017 and, in tree differencing this occurs often.

2018 The *Patch_{ST}* datatype has many interesting aspects that deserve some mention. First,
2019 by being globally synchronized – that is, explicit insertions and deletions with one hole –
2020 these patches are easy to merge. Moreover, we have seen that it is possible, and desirable,
2021 to encode patches as homogeneous types: a patch transform two values of the same
2022 member of the mutually recursive family.

2023 In conclusion, lacking an efficient *diff* algorithm meant that `stdiff` was an impor-
2024 tant step leading to new insights, but unfortunately was not worth pursuing further. This
2025 meant that a number of interesting topics such as the algebra of *Patch_{ST}* and the notion
2026 of cost for *Patch_{ST}* were abandoned indefinitely.

2027



2028

2029

PATTERN-EXPRESSION PATCHES

2030 The `stdiff` approach gave us a first representation of tree-structured patches over tree-
2031 structured data but was still deeply connected to edit-scripts: subtrees could only be
2032 copied once and could not be permuted. This means we still suffered from ambiguous
2033 patches, and, consequently, a computationally expensive *diff* algorithm. Overcoming
2034 the drawback of ambiguity requires a shift in perspective and abandoning edit-script
2035 based differencing algorithms. In this section we will explore the `hdiff` approach,
2036 where patches allow for trees to be arbitrarily permuted, duplicated or contracted (con-
2037 tractions are dual to duplications).

2038 Classical tree differencing algorithms start with the computation of tree matchings
2039 (Section 2.1.2), which identify the subtrees that should be copied. These tree matchings,
2040 however, must be restricted to order-preserving partial injections to be efficiently trans-
2041 lated to edit-scripts later. The `hdiff` approach never translates to edit-scripts, which
2042 means the tree matchings we compute are subject to *no* restrictions. In fact, `hdiff` uses
2043 these unrestricted tree matchings as *the patch*, instead of translating them *into* a patch.

2044 Suppose we want to describe a change that modifies the left element of a binary tree.
2045 If we had the full Haskell programming language available as the patch language, we
2046 could write something similar to function `c`, in Figure 5.1(A). Observing the function `c`
2047 we see it has a clear domain – a set of *Trees* that when applied to `c` yields a *Just* – which
2048 is specified by the pattern and guards. Then, for each tree in the domain we compute
2049 a corresponding tree in the codomain. The new tree is obtained from the old tree by
2050 replacing the `10` by `42` in-place. Closely inspecting this definition, we can interpret the
2051 matching of the pattern as a *deletion* phase and the construction of the resulting tree as a
2052 *insertion* phase. The `hdiff` approach represents the change in `c` exactly as that: a pattern



FIGURE 5.1: Haskell function and its graphical representation as a change. The change here modifies the left child of a binary node. Notation $\#_y$ is used to indicate y is a metavariable.

and a expression. Essentially, we write c as $\text{Chg } (\text{Bin } (\text{Leaf } 10) y) (\text{Bin } (\text{Leaf } 42) y)$ – represented graphically as in Figure 5.1(B).

With the added expressivity of referring to subtrees with metavariables we can represent more transformations than before. Take, for example, the change that swaps two subtrees – which cannot be written using an edit-script based approach – is given by $\text{Chg } (\text{Bin } x y) (\text{Bin } y x)$. Another helpful consequence of our design is that we effectively bypass the *choice* phase of the algorithm. When computing the differences between Bin Leaf Leaf and Leaf , for example, we do not have to chose one Leaf to copy because we can copy both with the help of a contraction operation, with a change such as: $\text{Chg } (\text{Bin } x x) x$. This aspect is crucial and enables us to write a linear *diff* algorithm.

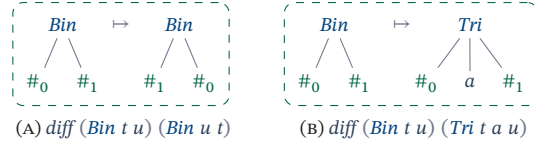
In this chapter we explore the representation and computation aspects of `hdiff`. The big shift in paradigm of `hdiff` also requires a more careful look into the metatheory and nuances of the algorithm, which were not present in our original paper [71]. The material in this chapter is developed from our ICFP’19 publication [71], shifting to the `generics-simplistic` library.

5.1 CHANGES

5.1.1 A CONCRETE EXAMPLE

Before exploring the generic implementation of our algorithm, let us look at a simple, concrete instance first, which sets the stage for the the generic implementation that will follow. Throughout this section we will explore the central ideas from our algorithm instantiated for a type of 2-3-trees:

```
data Tree = Leaf Int | Bin Tree Tree | Tri Tree Tree Tree
```

FIGURE 5.2: Illustration of two changes. Metavariables are denoted with $\#_x$.

The central concept of `hdiff` is the encoding of a *change*. Unlike previous work [51, 69, 46] which is based on tree-edit-distance [15] and hence uses only insertions, deletions and copies of the constructors encountered during the preorder traversal of a tree (Section 3.1.4), we go a step further. We explicitly model permutations, duplications and contractions of subtrees within our notion of *change*, where contraction here denotes the partial inverse of a duplication. The representation of a *change* between two values of type *Tree*, then, is given by identifying the bits and pieces that must be copied from source to destination making use of permutations and duplications where necessary.

A new datatype, *TreeC* φ , enables us to annotate a value of *Tree* with holes of type φ . Therefore, *TreeC Metavar* represents the type of *Tree* with holes carrying metavariables. These metavariables correspond to arbitrary trees that are *common subtrees* of both the source and destination of the change. These are exactly the bits that are being copied from the source to the destination tree. We refer to a value of *TreeC* as a *context*. For now, the metavariables will be simple *Int* values but later on they will need to carry additional information.

```

type Metavar = Int
data TreeC  $\varphi$  = Hole  $\varphi$ 
              | LeafC Int
              | BinC TreeC TreeC
              | TriC TreeC TreeC TreeC

```

A *change* in this setting is a pair of such contexts. The first context defines a pattern that binds some metavariables, called the deletion context; the second, called the insertion context, corresponds to the tree annotated with the metavariables that are supposed to be instantiated by the bindings given by the deletion context.

```

type Change  $\varphi$  = (TreeC  $\varphi$ , TreeC  $\varphi$ )

```

The change that transforms *Bin t u* into *Bin u t*, for example, is represented by a pair of *TreeC*, (*BinC* (*Hole* 0) (*Hole* 1), *BinC* (*Hole* 1) (*Hole* 0)), as seen in Figure 5.2. This change works on any tree built using the *Bin* constructor and swaps the children of

2099 the root. Note that it is impossible to define such swap operations in terms of insertions
 2100 and deletions—as used by most diff algorithms.

2101 5.1.1.1 APPLYING CHANGES

2102 Applying a change to a tree is done by unifying the metavariables in the deletion context
 2103 with said tree, and later instantiating the the insertion context with the obtained sub-
 2104 stitution. Later on, when we come to the generic setting, we will write the application
 2105 function using syntactic unification [89]. For this concrete example, we will continue
 2106 with the definition below.

2107 $chgApply :: Change\ Metavar \rightarrow Tree \rightarrow Maybe\ Tree$
 $chgApply\ (d, i)\ x = del\ d\ x \succcurlyeq ins\ i$

2108 Naturally, if the term x and the deletion context d are *incompatible*, this operation
 2109 will fail. Contrary to regular pattern-matching, we allow variables to appear more than
 2110 once on both the deletion and insertion contexts. Their semantics are dual: duplicate
 2111 variables in the deletion context must match equal trees, and are referred to as contrac-
 2112 tions, whereas duplicate variables in the insertion context will duplicate trees. Given a
 2113 deletion context ctx and source tree t , the del function tries to associate all the metavar-
 2114 ibles in the context with a subtree of the input $tree$. This can be done with standard
 2115 unification algorithms, as will be the case in the generic setting. Here, however, we use
 2116 a simple auxiliary function to do so.

2117 $del :: TreeC\ Metavar \rightarrow Tree \rightarrow Maybe\ (Map\ Metavar\ Tree)$
 $del\ ctx\ t = go\ ctx\ t\ empty$

2118 The go function, defined below, closely follows the structure of trees and contexts.
 2119 Only when we reach a *Hole* we check whether we have already instantiated the metavar-
 2120 ible stored there or not. If we encountered this metavariable before, we check that both
 2121 occurrences of the metavariable correspond to the same tree; if this is the first time we
 2122 encounter this metavariable, we instantiate the metavariable with the current tree.

2123 $go :: TreeC \rightarrow Tree \rightarrow Map\ Metavar\ Tree \rightarrow Maybe\ (Map\ Metavar\ Tree)$
 $go\ (LeafC\ n)\ (Leaf\ n')\ m = guard\ (n \equiv n') > return\ m$
 $go\ (BinC\ x\ y)\ (Bin\ a\ b)\ m = go\ x\ a\ m \succcurlyeq go\ y\ b$
 $go\ (TriC\ x\ y\ z)\ (Tri\ a\ b\ c)\ m = go\ x\ a\ m \succcurlyeq go\ y\ b \succcurlyeq go\ z\ c$
 $go\ (Hole\ i)\ t\ m = \text{case lookup } i\ m\ \text{of}$
 $\quad Nothing \rightarrow return\ (M.insert\ i\ t\ m)$
 $\quad Just\ t' \rightarrow guard\ (t \equiv t') > return\ m$
 $go\ -\ -\ m = Nothing$

2124 Once we have computed the substitution that unifies ctx and t , above, we instantiate
 2125 the variables in the insertion context with their respective values to obtain the resulting
 2126 tree. The ins function, defined below, performs this instantiation and fails only if the
 2127 change contains unbound variables.

```

ins :: TreeC Metavar → Map Metavar Tree → Maybe Tree
ins (LeafC n) m = return (Leaf n)
ins (BinC x y) m = Bin <$> ins x m <*> ins y m
ins (TriC x y z) m = Tri <$> ins x m <*> ins y m <*> ins z m
ins (Hole i) m = lookup i m
  
```

2129

2130

2131 5.1.1.2 COMPUTING CHANGES

2132 Next we will define the $chgTree$ function, which produces a change from a source and a
 2133 destination. Intuitively, the $chgTree$ function should try to exploit as many copy oppor-
 2134 tunities as possible. For now, we delegate the decision of whether a subtree should be
 2135 copied or not to an oracle: assume we have access to a function $wcs :: Tree \rightarrow Tree \rightarrow$
 2136 $Tree \rightarrow Maybe Metavar$, short for “which common subtree”. The call $wcs\ s\ d\ x$ returns
 2137 *Nothing* when x is not a subtree of s and d ; if x is a subtree of both s and d , it returns
 2138 *Just i*, for some metavariable i . The only condition we impose is injectivity of $wcs\ s\ d$:
 2139 that is, if $wcs\ s\ d\ x \equiv wcs\ s\ d\ y \equiv Just\ j$, then $x \equiv y$. In other words, equal metavariables
 2140 correspond to equal subtrees.

2141 There is an obvious inefficient implementation for wcs , that traverses both trees
 2142 searching for shared subtrees – hence postulating the existence of such an oracle is not a
 2143 particularly strong assumption to make. In Section 5.1.1.3, we provide an efficient imple-
 2144 mentation. For now, assuming the oracle exists allows for a clear separation of concerns.
 2145 The $chgTree$ function merely has to compute the deletion and insertion contexts, using
 2146 said oracle – the inner workings of the oracle are abstracted away cleanly.

```

chgTree :: Tree → Tree → Change Metavar
chgTree s d = let f = wcs s d
               in (extract f s, extract f d)
  
```

2148 The $extract$ function receives an oracle and a tree. It traverses its argument tree,
 2149 looking for opportunities to copy subtrees. It repeatedly consults the oracle, to determine
 2150 whether or not the current subtree should be shared across the source and destination.
 2151 If that is the case, we want our change to *copy* such subtree. That is, we return a *Hole*
 2152 whenever the second argument of $extract$ is a common subtree according to the oracle.

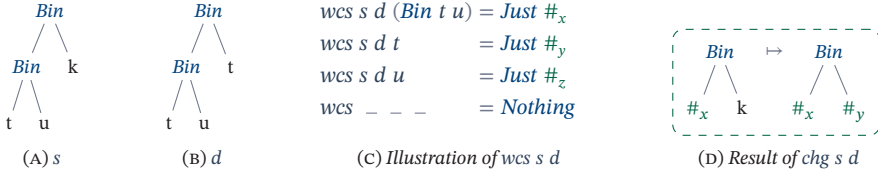


FIGURE 5.3: Context extraction must care to produce well-formed changes. The nested occurrence of t within $\text{Bin } t \ u$ here yields a change with an undefined variable on its insertion context.

2153 If the oracle returns *Nothing*, we move the topmost constructor to the context being
 2154 computed and recurse over the remaining subtrees.

```

extract :: (Tree → Maybe Metavar) → Tree → TreeC Metavar
extract o t = maybe (peel t) Hole (o t)
2155   where peel (Leaf n) = LeafC n
         peel (Bin a b) = BinC (extract o a) (extract o b)
         peel (Tri a b c) = TriC (extract o a) (extract o b) (extract o c)

```

2156 Note that if adopted a version of *wcs* that only returns a boolean value we would
 2157 not know what metavariable to use when a subtree is shared. Returning a value that
 2158 uniquely identifies a subtree allows us to keep the *extract* function linear in the number
 2159 of constructors in x (disregarding the calls to our oracle for the moment).

2160 This iteration of the *chgTree* function has a subtle bug, however. It does *not* produce
 2161 correct changes, that is, it is not the case that $\text{apply } (\text{chg } s \ d) \ s \equiv \text{Just } d$ for all s and
 2162 d . The problem can be observed when we pass a source and a destination tree where a
 2163 common subtree occurs by itself but also as a subtree of another common subtree. Such
 2164 situation is illustrated in Figure 5.3. In particular, the patch shown in Figure 5.3(D)
 2165 cannot be applied since the deletion context does not instantiate the metavariable $\#_y$,
 2166 which required by the insertion context.

2167 There are many ways to address the issue illustrated in Figure 5.3. We could replace
 2168 $\#_y$ by t and ignore the sharing or we could replace $\#_x$ by $\text{Bin } \#_y \ \#_z$, pushing the metavariables
 2169 to the leaves maximizing sharing. These would give rise to the changes shown in
 2170 Figure 5.4. There is a clear dichotomy between wanting to maximize the spine but at the
 2171 same time wanting to copy the larger trees, closer to the root. On the one hand, copies
 2172 closer to the root are intuitively easier to merge and less sharing means it is easier to
 2173 isolate changes to separate parts of the tree. On the other hand, sharing as much as possible
 2174 might capture the change being represented more closely.

2175 A third, perhaps less intuitive, solution to the problem in Figure 5.3 is to only share
 2176 uniquely occurring subtrees, effectively simulating the UNIX *diff* with the *patience*

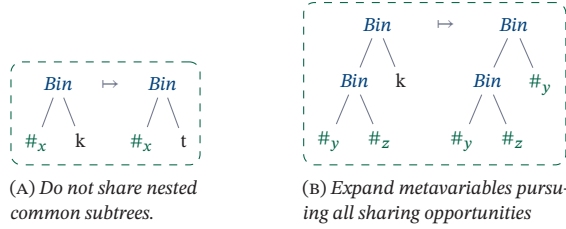


FIGURE 5.4: Two potential solutions to the problem of nested common subtrees, illustrated in Figure 5.3

option, which only copies uniquely occurring lines. In fact, to make this easy to experiment with, we will parameterize our final *extract* with which *context extraction mode* should be used to computing changes.

```

2180 data ExtractionMode = NoNested
                        | ProperShare
                        | Patience

```

The *NoNested* mode will forget sharing in favor of copying larger subtrees. It would drop the sharing of *t* producing Figure 5.4(A). The *ProperShare* mode is the opposite. It would produce Figure 5.4(B). Finally, *Patience* only share subtrees that occur only once in the source and once in the destination. For the inputs in Figure 5.3, extracting contexts under *Patience* mode would produce the same result as *NoNested*, but they are not the same in general. In fact, Figure 5.5 illustrates the changes that would be extracted following each *ExtractionMode* for the same source and destination.

In short, the *extract* function receives the *sharing map* and extracts the deletion and insertion context making up the change, caring that the produced change is well-scoped. We will give the final *extract* function when we get to its generic implementation. For the time being, let us move on to the intuition behind computing the *wcs* function efficiently for the concrete case of the *Tree* datatype.

5.1.1.3 DEFINING THE ORACLE FOR *Tree*

In order to have a working version of our differencing algorithm for *Tree* we must provide the *wcs* implementation, with type $Tree \rightarrow Tree \rightarrow Tree \rightarrow Maybe\ Metavar$. Given a fixed *s* and *d*, *wcs s d x* returns *Just i* if *x* is the *i*th subtree of *s* and *d* and *Nothing* if *x* does not appear in *s* or *d*. One implementation of this function computes the intersection of all

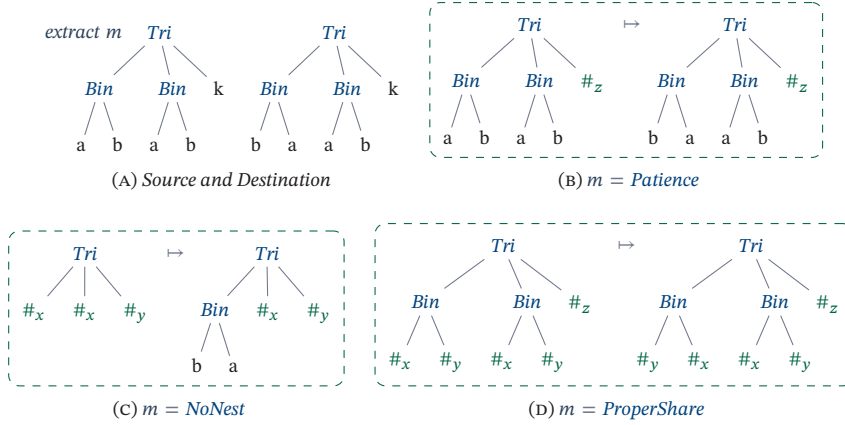


FIGURE 5.5: Different extraction methods for the same pair or trees.

the subtrees in s and d , and then search for the subtree x the resulting list. Enumerating all the subtrees of any *Tree* is straightforward:

$\text{subtrees} :: \text{Tree} \rightarrow [\text{Tree}]$

It is now easy to implement the wcs function: we compute the intersection of all the subtrees of s and d and use this list to determine whether the argument tree occurs in both s and d . This check is done with elemIndex which returns the index of the element when it occurs in the list.

$\text{wcs} :: \text{Tree} \rightarrow \text{Tree} \rightarrow \text{Tree} \rightarrow \text{Maybe Metavar}$
 $\text{wcs } s \ d \ x = \text{elemIndex } x \ (\text{subtrees } s \cap \text{subtrees } d)$

This implementation, however, is not particularly efficient. The inefficiency comes from two places: firstly, checking trees for equality is linear in the size of the tree; furthermore, enumerating all subtrees is exponential. If we want our algorithm to be efficient we *must* have an amortized constant-time wcs .

Defining $\text{wcs } s \ d$ efficiently consists, firstly, of computing a set of trees which contains the subtrees of s and d , and secondly, in being able to efficiently query this set for membership. Symbolic manipulation software, such as Computer Algebra Systems, perform similar computations frequently and their performance is just as important. These systems often rely on a technique known as *hash-consing* [36, 30], which is part of the canon of programming folklore. Hash-consing arises as a means of *maximal sharing* of subtrees in memory and constant time comparison – two trees are equal if they are

2218 stored in the same memory location – but it is by far not limited to it. We will be using
 2219 a variant of *hash-consing* to define *wcs s d*.

2220 To efficiently compare trees for equality we will be using cryptographic hash func-
 2221 tions [64] to construct a fixed length bitstring that uniquely identifies a tree modulo hash
 2222 collisions. Said identifier will be the hash of the root of the tree, which will depend on
 2223 the hash of every subtree, much like a *merkle tree* [65]. Suppose we have a function
 2224 *merkleRoot* that computes some suitable identifier for every tree, we can compare trees
 2225 efficiently by comparing their associated identifiers:

```
2226 instance Eq Tree where
      t ≡ u = merkleRoot t ≡ merkleRoot u
```

2227 The definition of *merkleRoot* function is straightforward. It is important that we
 2228 use the *merkleRoot* of the parts of a *Tree* to compute the *merkleRoot* of the whole. This
 2229 construction, when coupled with a cryptographic hash function, call it *hash*, is what
 2230 guarantee injectivity modulo hash collisions.

```
merkleRoot :: Tree → Digest
merkleRoot (LeafH n) = hash (concat [ "1", encode n ])
2231 merkleRoot (Bin x y) = hash (concat [ "2", merkleRoot x, merkleRoot y ])
merkleRoot (Tri x y z) = hash (concat [ "3", merkleRoot x, merkleRoot y, merkleRoot z ])
```

2232 Note that although it is theoretically possible to have false positives, when using a
 2233 cryptographic hash function the chance of collision is negligible and hence, in practice,
 2234 they never happen [64]. Nonetheless, it would be easy to detect when a collision has
 2235 occurred in our algorithm; consequently, we chose to ignore this issue.

2236 Recall we are striving for a constant time comparison, but the (\equiv) definition compar-
 2237 ing merkle roots is still linear as it must recompute the *merkleRoot* on every comparison.
 2238 We fix this by caching the hash associated with every node of a *Tree*. This is done by the
 2239 *decorate* function, illustrated Figure 5.6.

```
type TreeH = (TreeH', Digest)
data TreeH' = LeafH Int
2240           | BinH TreeH TreeH
           | TriH TreeH TreeH TreeH
decorate :: Tree → TreeH
```

2241 We omit the implementation of *decorate* for now, even if it is straightforward. More-
 2242 over, a generic version is introduced in Section 5.1.4. The *TreeH* datatype carries round
 2243 the *merkleRoot* of its first component, hence, enabling us to define (\equiv) in constant time.

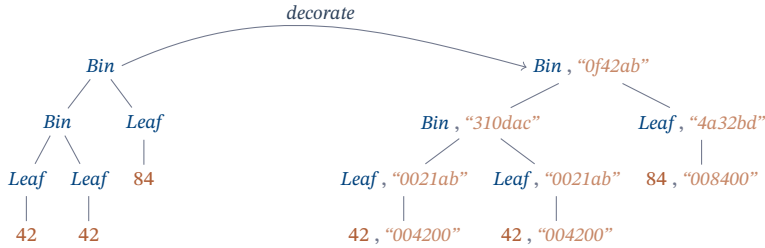


FIGURE 5.6: Example of decorating a tree with hashes, through the `decorate` function.

The second source of inefficiency, enumerating all possible subtrees, can be addressed by choosing a better data structure. To check whether a tree x is a subtree of a fixed s and d , it suffices to check whether the merkle root of x appears in a “database” of the common merkle roots of s and d . Given that a *Digest* is just a [*Word*], the optimal choice for such “database” is a Trie [18] mapping a [*Word*] to a *Metavar*. Trie lookups are efficient and hardly depend on the number of elements in the trie. In fact, our lookups run in amortized constant time here, as the length of a *Digest* is fixed.

Finally, we are able to write our efficient *wcs* oracle that concludes the implementation of our algorithm for the concrete *Tree* type. The *wcs* oracle will now receive *TreeH*, i.e., trees annotated with their merkle roots at every node, and will populate the “database” of common digests.

```

wcs :: TreeH → TreeH → TreeH → Maybe Metavar
wcs s d = lookup (mkTrie s ∩ mkTrie d) ∘ merkleRoot
2255   where (∩) :: Trie k v → Trie k u → Trie k v
        lookup :: Trie k v → [k] → Maybe v
        mkTrie :: TreeH → TreeH → Trie Word Metavar

```

The use of cryptographic hashes is unsurprising. They are almost folklore for speeding up a variety of computations. It is important to note that the efficiency of the algorithm comes from the novel representation of patches combined with an amortized constant time *wcs* function. Without being able to duplicate or permute subtrees, the algorithm would have to backtrack in a number of situations.

2261 5.1.2 REPRESENTING CHANGES GENERICALLY

Having seen how *FreeC* played a crucial role in defining changes for the *Tree* datatype, we continue with its generic implementation. In this section, we generalize the construction of *contexts* to any datatype supported by the `generics-simplistic` library.

Recall that a *context* over a datatype T is just a value of T augmented with an additional constructor used to represent *holes*. This can be done with the *free monad* construction provided by the `generics-simplistic` library: `HolesAnn κ fam ann h` datatype (Section 3.2.3) is a free monad in h . We recall its definition ignoring annotations below.

```

data Holes  $\kappa$  fam h a where
  Hole  ::                               h a  $\rightarrow$  Holes  $\kappa$  fam h a
  Prim  :: (PrimCnstr  $\kappa$  fam a)  $\Rightarrow$  a  $\rightarrow$  Holes  $\kappa$  fam h a
  Roll  :: (CompoundCnstr  $\kappa$  fam a)  $\Rightarrow$  SRep (Holes  $\kappa$  fam h) (Rep a)  $\rightarrow$  Holes  $\kappa$  fam h a

```

The *TreeC Metavar* datatype, defined in Section 5.1.1 to represent a value of type *Tree* augmented with metavariables is isomorphic to `Holes '[Int] '[Tree] (Const Int)`. Abstracting over the specific family for *Tree*, the datatype `Holes κ fam (Const Int)` gives a functor mapping an element of the family into its representation augmented with integers, which represent metavariables. But in this generic setting, it does not yet enable us to infer whether a metavariable matches over an opaque type or a recursive position, which will come to be important soon. Consequently, we will keep the information about whether the metavariable matches over an opaque value or not:

```

data Metavar  $\kappa$  fam at where
  # $\kappa$  :: (PrimCnstr  $\kappa$  fam at)
       $\Rightarrow$  Int  $\rightarrow$  Metavar  $\kappa$  fam at
  # $\kappa$  fam :: (CompoundCnstr  $\kappa$  fam at)
       $\Rightarrow$  Int  $\rightarrow$  Metavar  $\kappa$  fam at

```

With *Metavar* above, we can always retrieve the *Int* identifying the metavar, with the `metavarGet` function, but we maintain all the type-level information we may need to inspect at run-time. The *HolesMV* datatype below is convenient since most of the times our *Holes* structures will contain metavariables.

```

metavarGet :: Metavar  $\kappa$  fam at  $\rightarrow$  Int
type HolesMV  $\kappa$  fam = Holes  $\kappa$  fam (Metavar  $\kappa$  fam)

```

A *change* consists of a pair of a deletion context and an insertion context for the same type. These contexts are values of the mutually recursive family in question, augmented with metavariables.

```

data Chg  $\kappa$  fam at = Chg { .del :: HolesMV  $\kappa$  fam at
                          , .ins :: HolesMV  $\kappa$  fam at
                          }

```

2289 Applying a generic change c to an element x consists in unifying x with c_{del} , yield-
 2290 ing a substitution σ which can be applied to c_{ins} . This provides the usual denotational
 2291 semantics of changes as partial functions.

```

2292   chgApply :: (All Eq  $\kappa$ )  $\Rightarrow$  Chg  $\kappa$  fam at  $\rightarrow$  SFix  $\kappa$  fam at  $\rightarrow$  Maybe (SFix  $\kappa$  fam at)
   chgApply (Chg d i) x = either (const Nothing) (holesMapM uninstantiateHole  $\circ$  flip substApply i)
                        (unify d (sfixToHoles x))
   where uninstantiateHole _ = error "uninstantiated hole: (Chg d i) not well-scoped!"

```

2293 In a call to `chgApply c x`, since x has no holes, a successful unification means σ
 2294 assigns a term (no holes) for each metavariable in c_{del} . In turn, when applying σ to
 2295 c_{ins} we must guarantee that every metavariable in c_{ins} gets substituted, yielding a term
 2296 with no holes as a result. Attempting to apply a non-well-scoped change is a violation
 2297 of the contract of `applyChg`. We throw an error in that case and distinguish it from a
 2298 change c not being able to be applied to x because x is not an element of the domain
 2299 of c . The `uninstantiateHole` above will be called in the precise situation where holes were left
 2300 uninstantiated in `substApply σ c_{ins}`

2301 In general, we expect a value of type `Chg` to be well-scoped, that is, all the variables
 2302 that are present in the insertion context must also occur on the deletion context, in
 2303 Haskell:

```

2304   vars      :: HolesMV  $\kappa$  fam at  $\rightarrow$  Map Int Arity
   wellscoped :: Chg  $\kappa$  fam at  $\rightarrow$  Bool
   wellscoped (Chg d i) = keys (vars i)  $\equiv$  keys (vars d)

```

2305 A `Chg` is very similar to a *tree matching* (Section 2.1.2) with less restrictions. In
 2306 other words, it arbitrarily maps subtrees from the source to the destination. From an
 2307 algebraic point of view, this already gives us a desirable structure, as we will explore next
 2308 in Section 5.1.3. In fact, we argue that there is no need to translate the tree matching into
 2309 an edit-script, like most traditional algorithms do. The tree matching should be used as
 2310 the representation of change.

2311 5.1.3 META THEORY

2312 In this section we will look into how `Chg` admits a simple composition operation which
 2313 makes a partial monoid. Through the remainder of this section we will assume changes
 2314 have all been α -converted to never capture names and denote the application function of
 2315 a change, `applyChg c`, as $\llbracket c \rrbracket$. We will also abuse notation and denote `substApply σ p` by
 2316 σp , whenever the context makes it clear that σ is a substitution. Finally, we will abide by
 2317 the Barendregt convention [12] in our proofs and metatheory – that is, all changes that
 2318 appear in in some mathematical context have their bound variable names independent
 2319 of each other, to put it differently, no two changes will accidentally share a variable name.

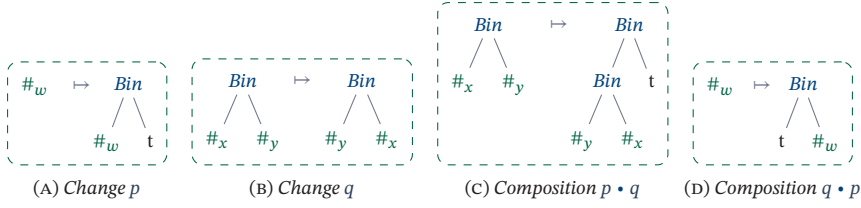


FIGURE 5.7: Example of change composition. The composition usually can be applied to less elements than its parts and is clearly not commutative.

The composition of two changes, say, p after q , returns a change that maps a subset of the domain of q into a subset of the image of p . Figure 5.7, for example, illustrates two changes and their two different compositions. In the case of Figure 5.7 both $p \bullet q$ and $q \bullet p$ exist, but this is not the case generally. The composition of two changes $p \bullet q$ is defined if and only if the image of $\llbracket q \rrbracket$ has elements in common with the domain of $\llbracket p \rrbracket$. In other words, when q_{ins} is unifiable with p_{del} . In fact, let $\sigma = \text{unify } q_{\text{ins}} \ p_{\text{del}}$, the composition $p \bullet q$ is given by $\text{Chg } (\sigma \ q_{\text{del}}) (\sigma \ p_{\text{ins}})$.

(\bullet) :: $\text{Chg } \kappa \text{ fam } at \rightarrow \text{Chg } \kappa \text{ fam } at \rightarrow \text{Maybe } (\text{Chg } \kappa \text{ fam } at)$
 $p \bullet q = \text{case } \text{unify } p_{\text{del}} \ q_{\text{ins}} \text{ of}$
 Left $_ \rightarrow \text{Nothing}$
 Right $\sigma \rightarrow \text{Just } (\text{Chg } (\text{substApply } \sigma \ q_{\text{del}}) (\text{substApply } \sigma \ p_{\text{ins}}))$

Note that it is inherent that purely structural composition of two changes p after q yields a change, $p \bullet q$, that potentially misses sharing opportunities. Imagine that p inserts a subtree t that was deleted by q . Our composition algorithm posesses no information that this t is to be treated as a copy. This also occurs in the edit-script universe: composing patches yields worse patches than recomputing differences. We can imagine that a more complicated composition algorithm might be able to recover the copies in those situations.

We do not particularly care whether composition produces *the best* change possible or not. We do not even have a notion of *best* at the moment. It is vital, however, that it produces a correct change. That is, the composition of two patches is indistinguishable from the composition of their application functions.

Lemma 5.1.1 (Composition Correct). *For any changes p and q and trees x and y aptly typed; we have $\llbracket p \bullet q \rrbracket x \equiv \text{Just } y$ if and only if $\exists z. \llbracket q \rrbracket x \equiv \text{Just } z \wedge \llbracket p \rrbracket z \equiv \text{Just } y$.*

Proof. **if.** Assuming $\llbracket p \bullet q \rrbracket x \equiv \text{Just } y$, we want to prove there exists z such that $\llbracket q \rrbracket x \equiv \text{Just } z$ and $\llbracket p \rrbracket z \equiv \text{Just } y$. Let σ be the result of $\text{unify } p_{\text{del}} \ q_{\text{ins}}$, witnessing $p \bullet q$; let γ be the result of $\text{unify } (\sigma \ q_{\text{del}}) \ x$, witnessing the application.

Take $z = (\gamma \circ \sigma) q_{\text{ins}}$, and let us prove $\gamma \circ \sigma$ unifies p_{del} and z .

$$\begin{aligned}
 & (\gamma \circ \sigma) p_{\text{del}} \equiv (\gamma \circ \sigma) z && \{z \text{ has no variables}\} \\
 \iff & (\gamma \circ \sigma) p_{\text{del}} \equiv z && \{\text{definition of } z\} \\
 \iff & (\gamma (\sigma p_{\text{del}}) \equiv \gamma (\sigma q_{\text{ins}})) && \{\text{hypothesis}\} \\
 \iff & \sigma p_{\text{del}} \equiv \sigma q_{\text{ins}}
 \end{aligned}$$

2344 Hence, p can be applied to z , and the result is $(\gamma \circ \sigma) p_{\text{ins}}$, which by hypothesis, is
 2345 equal to y .

2346 **only if.** Assuming there exists z such that $\llbracket q \rrbracket x \equiv \text{Just } z$ and $\llbracket p \rrbracket z \equiv \text{Just } y$, we want
 2347 to prove that $\llbracket p \bullet q \rrbracket x \equiv \text{Just } y$. Let α be such that $\alpha q_{\text{del}} \equiv x$, hence, $z \equiv \alpha q_{\text{ins}}$;
 2348 Let β be such that $\beta p_{\text{del}} \equiv z$, hence $y \equiv \beta p_{\text{ins}}$.

- a) First we prove that $p \bullet q$ is defined, that is, there exists σ' that unifies q_{ins} and p_{del} . Recall α and β have disjoint variables because we assume p and q have a disjoint set of names. Let $\sigma' = \alpha \cup \beta$, which corresponds to $\alpha \circ \beta$ or $\beta \circ \alpha$ because of they have disjoint set of names.

$$\begin{aligned}
 & \sigma' q_{\text{ins}} \equiv \sigma' p_{\text{del}} && \{\text{disjoint supports}\} \\
 \iff & \alpha q_{\text{ins}} \equiv \beta p_{\text{del}} && \{\text{definition of } z\} \\
 \iff & z \equiv \beta p_{\text{del}}
 \end{aligned}$$

2349 Since σ' unifies q_{ins} and p_{del} , let σ be their *most general unifier*. Then, $\sigma' \equiv$
 2350 $\gamma \circ \sigma$ for some γ and $p \bullet q \equiv \text{Chg } (\sigma q_{\text{del}}) (\sigma p_{\text{ins}})$.

- b) Next we prove $\llbracket p \bullet q \rrbracket x \equiv \text{Just } y$. First we prove σq_{del} unifies with x .

$$\begin{aligned}
 & x \equiv \beta q_{\text{del}} && \{\text{Disj. supports; Def. } \sigma'\} \\
 \iff & x \equiv \gamma (\sigma q_{\text{del}}) && \{x \text{ has no variables}\} \\
 \iff & \gamma x \equiv \gamma (\sigma q_{\text{del}})
 \end{aligned}$$

Hence, $\llbracket p \bullet q \rrbracket x$ evaluates to $\gamma (\sigma p_{\text{ins}})$. Proving it coincides with y is a straightforward calculation:

$$\begin{aligned}
 & \gamma (\sigma p_{\text{ins}}) \equiv y && \{\text{Def. } y\} \\
 \iff & \gamma (\sigma p_{\text{ins}}) \equiv \alpha p_{\text{ins}} && \{\text{Disj. supports; Def. } \sigma'\} \\
 \iff & \gamma (\sigma p_{\text{ins}}) \equiv \gamma (\sigma p_{\text{ins}})
 \end{aligned}$$

2351

2352

□

2353 Once we have established that composition is correct with respect to application, we
 2354 would like to ensure composition is associative. But first we need to specify what we
 2355 mean by *equal* changes. We will consider an extensional equality over changes. Two
 2356 changes are said to be equivalent if and only if they are indistinguishable through their
 2357 application semantics.

2358 **Definition 5.1.1** (Change Equivalent). Two changes p and q are said to be equivalent,
 2359 denoted $p \approx q$, if and only if $\forall x . \llbracket p \rrbracket x \equiv \llbracket q \rrbracket x$

2360 **Lemma 5.1.2** (Definability of Composition). *Let p, q and r be aptly typed changes, then,*
 2361 *$(p \bullet q) \bullet r$ is defined if and only if $p \bullet (q \bullet r)$ is defined.*

Proof. if. Assuming $(p \bullet q) \bullet r$ is defined, Let σ and θ be such that $\sigma p_{\text{del}} \equiv \sigma q_{\text{ins}}$ and
 $\theta (\sigma q_{\text{del}}) \equiv \theta r_{\text{ins}}$. We must prove that (a) r_{ins} unifies with q_{del} through some
 substitution θ' and (b) $\sigma' q_{\text{ins}}$ unifies with p_{del} . Take $\theta' = \theta \circ \sigma$, then:

$$\begin{aligned} & (\theta \circ \sigma) r_{\text{ins}} \equiv (\theta \circ \sigma) q_{\text{del}} & \{ \text{support } \sigma \cap \text{vars } r \equiv \emptyset \} \\ \iff & \theta r_{\text{ins}} \equiv (\theta \circ \sigma) q_{\text{del}} \end{aligned}$$

2362 Let ζ be the idempotent *most general unifier* of r_{ins} and q_{del} , it follows that $\theta' = \gamma \circ \zeta$
 2363 for some γ . Consequently, $q \bullet r = \text{Chg } (\zeta r_{\text{del}}) (\zeta q_{\text{ins}})$.

Now, we must construct σ' to unify p_{del} and ζq_{ins} , which enables the construction
 of $p \bullet (q \bullet r)$. Let $\sigma' = \theta \circ \sigma$ and reduce it to one of our assumptions:

$$\begin{aligned} & \theta (\sigma p_{\text{del}}) \equiv \theta (\sigma (\zeta q_{\text{ins}})) & \{ \theta \circ \sigma \equiv \gamma \circ \zeta \} \\ \iff & \theta (\sigma p_{\text{del}}) \equiv \gamma (\zeta (\zeta q_{\text{ins}})) & \{ \zeta \text{ idempotent} \} \\ \iff & \theta (\sigma p_{\text{del}}) \equiv \gamma (\zeta q_{\text{ins}}) & \{ \theta \circ \sigma \equiv \gamma \circ \zeta \} \\ \iff & \theta (\sigma p_{\text{del}}) \equiv \theta (\sigma q_{\text{ins}}) \\ \iff & \sigma p_{\text{del}} \equiv \sigma q_{\text{ins}} \end{aligned}$$

2364

2365 **only if.** Analogous.

2366

□

2367 **Lemma 5.1.3** (Associativity of Composition). *Let p, q and r be aptly typed changes such*
 2368 *that $(p \bullet q) \bullet r$ is defined, then $(p \bullet q) \bullet r \approx p \bullet (q \bullet r)$.*

2369 *Proof.* Straightforward application of Lemma 5.1.2 and Lemma 5.1.1. □

2370 **Lemma 5.1.4** (Identity of Composition). *Let p be a change, then $\epsilon = \text{Chg } \#_x \#_x$ is the*
 2371 *identity of composition. That is, $p \bullet \epsilon \approx p \approx \epsilon \bullet p$.*

2372 *Proof.* Trivial; ϵ unifies with all possible terms. □

2373 Lemmas 5.1.3 and 5.1.4 establish a partial monoid structure for Chg and \bullet under
 2374 extensional change equality, \approx . As we shall see next, however, it is not trivial to squeeze
 2375 more structure out of this change representation. \triangleleft *I would have enjoyed to be able to*
 2376 *spend more time studying the metatheory. Obviously, it is not because the options discussed*
 2377 *next failed that there exists no options to extend the metatheory whatsoever. It is still worth*
 2378 *discussing the difficulties I encountered while trying to use standard techniques, below.* \triangleright

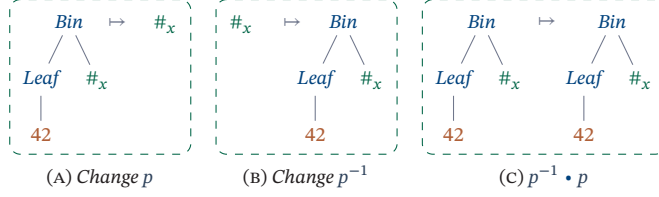


FIGURE 5.8: Example of a change, its inverse and their composition

2379 LOOSE ENDS. The first thing that comes to mind is the definition of the inverse of a
 2380 change. Since changes are well-scoped, that is, $\text{vars } p_{\text{del}} \equiv \text{vars } p_{\text{ins}}$ for any change p ,
 2381 defining the inverse of a change p , denoted p^{-1} , is trivial:

$$\begin{aligned} & \cdot^{-1} :: \text{Chg } \kappa \text{ fam } at \rightarrow \text{Chg } \kappa \text{ fam } at \\ 2382 \quad p^{-1} &= \text{Chg } p_{\text{ins}} p_{\text{del}} \end{aligned}$$

2383 Naturally, then, we would expect that $p \cdot p^{-1} \approx \epsilon$, but that is not the case. The
 2384 domain of ϵ is the entire set of trees, but the domain of $p \cdot p^{-1}$ is generally strictly smaller.
 2385 Consequently, we can easily find a tree t such that $\llbracket \epsilon \rrbracket t \equiv \text{Just } t$ but $\llbracket p \cdot p^{-1} \rrbracket \equiv \text{Nothing}$.
 2386 Take, for example, the change shown in Figure 5.8.

2387 The problem with inverses above stems from $p \cdot p^{-1}$ being *less general* than the
 2388 identity, since it has a smaller domain. In other words, $p \cdot p^{-1}$ works on a subset of
 2389 the domain of ϵ , but it performs the same action as ϵ for the elements it is defined. It
 2390 is natural then to attempt to talk about changes modulo their domain. We could think
 2391 of stating $p \leq q$ whenever $\llbracket p \rrbracket \subseteq \llbracket q \rrbracket$. That is, when p and q are the same except that
 2392 the domain of q is larger. This \leq is known as the usual *extension order* [88], and when
 2393 instantiated for our particular case, yields the definition below.

2394 **Definition 5.1.2** (Extension Order). Let p and q be two aptly typed changes; we say that
 2395 q is an extension of p , denoted $p \leq q$, if and only if $\forall x \in \text{dom } p \cdot \llbracket p \rrbracket x \equiv \llbracket q \rrbracket x$. In
 2396 other words, $p \leq q$ when q coincides with p in a restriction of its domain.

2397 This gives us a partial order on changes and it is the case that $p \cdot p^{-1} \leq \epsilon$ and
 2398 $p^{-1} \cdot p \leq \epsilon$. Attempting to identify $p \cdot p^{-1}$ as somehow equivalent to ϵ using \leq will not
 2399 work, however. We could think of defining a notion of *approximate changes*, denoted
 2400 $p \sim q$, by whether p and q are comparable under \leq . This would not yield an equivalence
 2401 relation since \sim is not transitive, as illustrated in Figure 5.9). Moreover, the extension
 2402 order cannot be used to define the *best* change between two elements x and y . Take x to
 2403 be $\text{Bin } (\text{Bin } a \ b) \ a$ and y to be $\text{Bin } (\text{Bin } b \ a) \ a$, for which two uncomparable candidate
 2404 changes are shown in Figure 5.10.

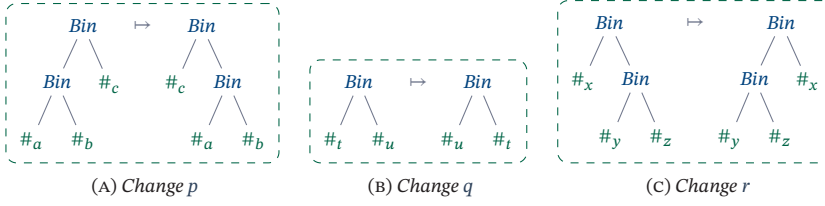


FIGURE 5.9: Three changes such that $p \sim q$ (because $p \leq q$) and $q \sim r$ (because $r \leq q$). Yet, $p \not\sim r$ since its not the case that $r \leq p$ or $p \leq r$ holds.

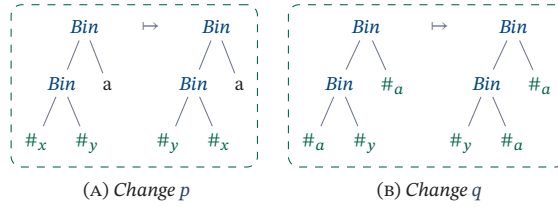


FIGURE 5.10: Two changes that could be used to transform the same element x but are not comparable under the extension order (\leq).

2405 This short discussion does not mean that there is *no* suitable way to compare the
 2406 changes in Figure 5.10 or to define \sim in such a way that the changes in Figure 5.9 can
 2407 be considered equivalent. It does mean, however, that simply comparing the domain of
 2408 changes is a weak definition and a robust definition will probably be significantly more
 2409 involved.

2410 5.1.4 COMPUTING CHANGES

2411 Having seen how *Chg* has the basic properties we would expect, we move on to com-
 2412 puting them. In this section we define the generic counterpart to the *chgTree* function
 2413 (Section 5.1.1). Recall that the differencing algorithm starts by computing the *sharing*
 2414 *map* of its source s and destination d , which enable us to efficiently decide if a given tree
 2415 x is a subtree of s and d . Later, we use this sharing map and *extract* the deletion and
 2416 insertion contexts, according to some extraction mode, which ensure we will produce
 2417 well-scoped changes (Figure 5.4).

2418 **data** *ExtractionMode* = *NoNested* | *ProperShare* | *Patience*

2419 The *sharing map* is central to the efficiency of the differencing algorithm, but it
 2420 marks subtrees for sharing regardless of underlying semantics, which can be a prob-
 2421 lem when the trees in question represent complex structures such as abstract syntax
 2422 trees. We must be careful not to *overshare* trees. Imagine a local variable declaration
 2423 `int x = 0;` inside an arbitrary function. This declaration should *not* be shared with
 2424 another syntactically equal declaration in another function. A careful analysis of what
 2425 can and cannot be shared would require domain-specific knowledge of the programming
 2426 language in question. Nevertheless, we can impose different restrictions that make it
 2427 *unlikely* that values will be shared across scope boundaries. A simple and effective such
 2428 measure is not sharing subtrees with height strictly less than one (or a configurable pa-
 2429 rameter). This keeps constants and most variable declarations from being shared, effec-
 2430 tively avoiding the issue. \triangleleft I would like to reiterate the avoiding-the-issue aspect of this
 2431 decision. I did attempt to overcome this with a few methods which will be discussed later
 2432 (Section 5.4). None of my attempts at solving the issue were successful, hence, the best op-
 2433 tion really became avoiding the issue by making sure that we can easily exclude certain
 2434 trees from being shared. \triangleright

2435 5.1.4.1 WHICH COMMON SUBTREE, GENERICALLY

2436 Similarly to example from Section 5.1.1, the first thing we must do is to annotate our
 2437 trees with hashes at every point. The *Holes* datatype from `generics-simplistic`
 2438 also supports annotations. Unlike the concrete example, however, we will also keep
 2439 the height of each tree to enable us to easily forbid sharing trees smaller than a certain
 2440 height. The *PrepFix* datatype, defined below, serves the same purpose as the simpler
 2441 *TreeH*, from our concrete example.

```
2442 data PrepData a = PrepData {getDigest :: Digest, getHeight :: Int}
type PrepFix  $\kappa$  fam = SFixAnn  $\kappa$  fam PrepData
```

2443 The *decorate* function can be written with the help of synthesized attributes (Sec-
 2444 tion 3.2.3.1). The homonym *synthesize* function from `generics-simplistic` serves
 2445 this very purpose. We omit the algebra passed to *synthesize* but invite the interested
 2446 reader to check *Data.HDiff.Diff.Preprocess* in the source (Appendix A).

```
2447 decorate :: (All Digestible  $\kappa$ )  $\Rightarrow$  SFix  $\kappa$  fam at  $\rightarrow$  PrepFix  $\kappa$  fam at
decorate = synthesize ...
```

2448 The algebra used by *decorate*, above, computes a hash at each constructor of the tree.
 2449 The hashes are computed from the a unique identifier per constructor and a concatena-
 2450 tion of the hashes of the subtrees. The hash of the root in Figure 5.6, for example, is
 2451 computed with a call to *hash* (*concat* [*“Main.Tree.Bin”*, *“310dac”*, *“4a32bd”*])). This
 2452 ensures that hashes uniquely identify a subtree modulo hash collisions.

2453 After preprocessing the input trees we traverse them and insert every hash we see
 2454 in a hash map from hashes to integers. These integers count how many times we have
 2455 seen a tree, indicating the arity of a subtree. Shared subtrees occur with arity of at least
 2456 two: once in the deletion context and once in the insertion context. The underlying
 2457 datastructure is a *Int64*-indexed trie [18] as our datastructure. \triangleleft *I would like to also*
 2458 *implemented this algorithm with a big-endian Patricia Tree [78] and compare the results.*
 2459 *I think the difference would be small, but worth considering when working on a production*
 2460 *implementation* \triangleright .

```
2461 type Arity = Int
      buildArityMap :: PrepFix a  $\kappa$  fam ix  $\rightarrow$  Trie Arity
```

2462 A call to *buildArityMap* with the annotated tree shown in Figure 5.6, for example,
 2463 would yield the map *fromList* [(*"0f42ab"*, 1), (*"310dac"*, 1), (*"0021ab"*, 2), ...].

2464 After processing the *arity* maps for both the source tree and destination tree, we con-
 2465 struct the *sharing* map, which consists in the intersection of the arity maps and a final
 2466 pass adding a unique identifier to every key. We also keep track of how many metavariables
 2467 were assigned, so we can always allocate fresh names without having to go inspect
 2468 the whole map again. This is just a technical consequence of working with binders ex-
 2469 plicitly.

```
      type MetavarAndArity = MAA {getMetavar :: Int, getArity :: Arity}
      buildSharingMap :: PrepFix a  $\kappa$  fam ix  $\rightarrow$  PrepFix a  $\kappa$  fam ix
                         $\rightarrow$  (Int, Trie MetavarAndArity)
      buildSharingMap x y = T.mapAccum ( $\lambda i ar \rightarrow (i + 1, MAA i ar)$ ) 0
                        $ T.zipWith (+) (buildArityMap x) (buildArityMap y)
```

2471 The final *wcs s d* is straightforward: we preprocess the trees with their hash and
 2472 height then compute their sharing map, which is used to lookup the common subtrees.
 2473 Yet, the whole point of preprocessing the trees was to avoid the unnecessary recomputa-
 2474 tion of their hashes. Consequently, we are better off carrying these preprocessed trees
 2475 everywhere through the computation of changes. The final *wcs* function will have its
 2476 type slightly adjusted and is defined below.

```
      wcs :: (All Digestible  $\kappa$ )  $\Rightarrow$  PrepFix  $\kappa$  fam at  $\rightarrow$  PrepFix  $\kappa$  fam at
                         $\rightarrow$  PrepFix  $\kappa$  fam at  $\rightarrow$  Maybe Int
2477      wcs s d = let m = buildSharingMap s d
                  in famp getMetavar  $\circ$  flip T.lookup m  $\circ$  getDigest  $\circ$  getAnnot
```

2478 Let $f = wcs s d$ for some s and d . Computing f itself is linear and takes $\mathcal{O}(n + m)$
 2479 time, where n and m are the number of constructors in s and d . A call to $f x$ for some x ,
 2480 however, is answered in $\mathcal{O}(1)$ due to the bounded depth of the patricia tree.

2481 We chose to use a cryptographic hash function [64] and ignore the remote possibil-
 2482 ity of hash collisions. Although it would not be hard to detect these collisions whilst
 2483 computing the arity map, doing so would incur a performance penalty. Checking for
 2484 collisions would require us to store the tree with its associated hash instead of only stor-
 2485 ing the hash. Then, on every insertion we could check that the inserted tree matches
 2486 with the tree already in the map. \triangleleft *If I had used a non-cryptographic hash, which are*
 2487 *much faster to compute than cryptographic hash functions, I would have had to employ*
 2488 *the collision detection mechanism above. This would cost a significant amount of time. I*
 2489 *believe it is worth paying the price for a more expensive hash function.* \triangleright

2490 5.1.4.2 CONTEXT EXTRACTION

2491 After computing the set of common subtrees, we must decide which of those subtrees
 2492 should be shared. Shared subtrees are abstracted by a metavariable in every location
 2493 they would occur at in the deletion and insertion contexts.

2494 Recall that we chose to never share subtrees with height smaller than a given pa-
 2495 rameter. Our choice is very pragmatic in the sense that we can preprocess the necessary
 2496 information and it effectively avoids most of the oversharing without involving domain
 2497 specific knowledge. The *CanShare* below is a synonym for a predicate over trees used
 2498 to decide whether we can share a given tree or not.

2499 **type** *CanShare* κ fam = $\forall ix \ . \text{ PrepFix } \kappa \text{ fam } ix \rightarrow \text{Bool}$

2500 The *extract* function takes an *ExtractionMode*, a sharing map and a *CanShare* predi-
 2501 cate and two preprocessed fixpoints to extract contexts from. The reason we receive two
 2502 trees at the same time and produce two contexts is because modes like *NoNested* perform
 2503 some cleanup that depends on global information.

2504
$$\begin{aligned} \text{extract} &:: \text{ExtractionMode} \rightarrow \text{CanShare } \kappa \text{ fam} \rightarrow \text{IsSharedMap} \\ &\rightarrow (\text{PrepFix } \kappa \text{ fam} \text{ :}^* \text{ :} \text{ PrepFix } \kappa \text{ fam}) \text{ at} \rightarrow \text{Chg } \kappa \text{ fam at} \end{aligned}$$

2505 \triangleleft *To some extent, we could compare context extraction to the translation of tree map-*
 2506 *plings into edit-scripts: our tree matching is encoded in wcs and instead of computing an*
 2507 *edit-scripts, we compute terms with metavariables. Classical algorithms are focused in*
 2508 *computing the least cost edit-script from a given tree mapping. In our case, the notion of*
 2509 *least cost hardly makes sense – besides not having defined a cost semantics to our changes,*
 2510 *we are interested in those that merge better which might not necessarily be those that in-*
 2511 *sert and delete the least amount of constructors. Consequently, there is a lot of freedom in*
 2512 *defining our context extraction techniques. We will look at three particular examples next,*
 2513 *but I sketch other possibilities later (Section 5.4).* \triangleright

2514 EXTRACTING WITH *NoNested*. Extracting contexts with the *NoNested* mode happens
 2515 in two passes. We first extract the contexts naively, then make a second pass removing
 2516 the variables that appear exclusively in the insertion. To keep the extraction algorithm
 2517 linear is important to *not* forget which common subtrees have been substituted on the
 2518 first pass. Hence, we create a context that contains metavariables and their associated
 2519 tree.

```

noNested1 :: CanShare κ fam → Trie MetavarAndArity → PrepFix κ fam at
           → Holes κ fam (Const Int ∙: PrepFix a κ fam) at
noNested1 h sm x@(PrimAnn _ xi) = Prim xi
noNested1 h sm x@(SFixAnn ann xi)
2520   = if h x then maybe recurse (mkHole x) $ lookup (getDigest ann) sm
           else recurse
where recurse = Roll (repMap (noNested1 h sm) xi)
mkHole x v = Hole (Const (getMetavar v) ∙: x)

```

2521 The second pass maps over the holes in the output from the first pass and decides
 2522 whether to transform the *Const Int* into a *Metavar κ fam* or whether to forget this was
 2523 a potential shared tree and keep the tree instead. We will omit the implementation of
 2524 the second pass. It consists in a straightforward traversal of the output of *noNested1*, we
 2525 direct the interested reader to check *Data.HDiff.Diff.Modes* in the source code for more
 2526 details (Appendix A).

2527 EXTRACTING WITH *Patience*. The *Patience* extraction can be done in a single pass. Un-
 2528 like *noNested1* above, instead of simply looking a hash up in the sharing map, it will
 2529 further check that the given hash occurs with arity two – indicating the tree in question
 2530 occurs once in the source tree and once in the destination. This completely bypasses the
 2531 issue with *NoNested* producing insertion contexts with undefined variables and requires
 2532 no further processing. The reason for it is that the variables produced will appear with
 2533 the same arity as the trees they abstract, and in this case, it will always be two: once in
 2534 the deletion context and once in the insertion context.

```

patience :: CanShare κ fam → Trie MetavarAndArity → PrepFix a κ fam at
           → Holes κ fam (Metavar κ fam) at
patience h sm x@(PrimAnn _ xi) = Prim xi
patience h sm x@(SFixAnn ann xi)
2535   = if h x then maybe recurse (mkHole x) $ lookup (getDigest ann) sm
           else recurse
where recurse = Roll (repMap (patience h sm) xi)
mkHole x v | getArity v ≡ 2 = Hole (#fam(getMetavar v))
           | otherwise      = sfixToHoles x

```

2536 EXTRACTING WITH *ProperShares*. The *ProperShares* method prefers sharing smaller
 2537 subtrees more times instead of but bigger subtrees, which might shadow nested com-
 2538 monly occurring subtrees (Figure 5.3).

2539 Given a source s and a destination d , we say that a tree x is a *proper-share* between
 2540 s and d whenever no subtree of x occurs in s and d with arity greater than that of x . In
 2541 other words, x is a proper-share if and only if all of its subtrees occur only as subtrees of
 2542 other occurrences of x . For the two trees below, u is a proper-share but *Bin* t u is not: t
 2543 occurs once *outside* *Bin* t u .



2545 Extracting contexts with under the *ProperShare* mode consists in annotating the
 2546 source and destination trees with a boolean indicating whether or not they are a proper
 2547 share, then proceeding just like *Patience*, but instead of checking that the arity must be
 2548 two, we check that the tree is classified as a *proper-share*. It is important to use anno-
 2549 tated fixpoints to maintain performance, but the code is very similar to the previous two
 2550 methods and, hence, omitted.

2551 THE *chg* FUNCTION. Finally, the generic *chg* function receives a source and destina-
 2552 tion trees, s and d , and computes a change that encodes the information necessary to
 2553 transform the source into the destination according to some extraction mode *extMode*.
 2554 In our prototype, the extraction mode comes from a command line option.

2555

```

chg :: (All Digestible  $\kappa$ )  $\Rightarrow$  SFix  $\kappa$  fam at  $\rightarrow$  SFix  $\kappa$  fam at  $\rightarrow$  Patch  $\kappa$  fam at
chg  $x$   $y$  = let  $dx$       = decorate  $x$ 
               $dy$       = decorate  $y$ 
              ( $-, sh$ ) = buildSharingMap opts  $dx$   $dy$ 
              in extract extMode canShare ( $dx$   $dy$ )

where
  canShare  $t$  = 1 < treeHeight (getConst (getAnn  $t$ ))
  
```

2556 5.2 THE TYPE OF PATCHES

2557 Up until now we have seen how *changes* consisting of a deletion and an insertion con-
 2558 text are a suitable representation for encoding transformations between trees. In fact,
 2559 changes are very similar to *tree matchings* (Section 2.1.2) but with fewer restrictions.

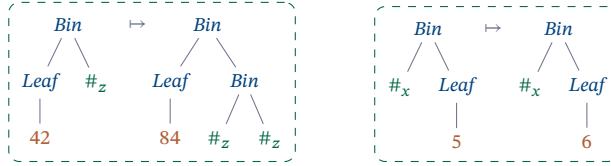


FIGURE 5.11: Example of disjoint changes. Each change is delimited by a dashed box. The leftmost change modifies the left child and duplicates the right child without changing its content. The rightmost change operates solely on the right child.

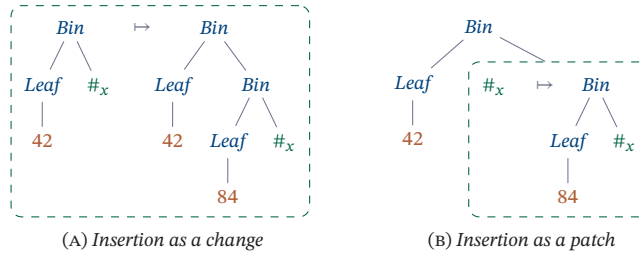


FIGURE 5.12: A change with redundant information on the left and its minimal representation on the right, with an evident spine.

From a synchronization point of view, however, these *changes* are very difficult to merge. They do not explicitly encode enough information for that.

Synchronizing changes requires us to understand which constructors in the deletion context are, in fact, just being copied over in the insertion context. Take Figure 5.11, where one change operates exclusively on the right child of a binary tree whereas the other alters the left child and duplicates the right child in-place. These changes are clearly *disjoint*, since they modify the content of different subtrees of the source. Consequently it should be possible to be automatically synchronize them. To recognize them as *disjoint* changes, though will require more information than what is provided by *Chg*.

Observing the definition of *Chg* reveals that the deletion context might *delete* many constructors that that are being inserted, in the same place, by the insertion context. The changes from Figure 5.11, for example, conceal the fact that the *Bin* at the root of the source tree is, in fact, being copied in both changes. Following the *stdiff* nomenclature, the *Bin* at the root of both changes in Figure 5.11 should be places in the *spine* of the patch. That is, it is copied over from source to destination but it leads to changes further down the tree.

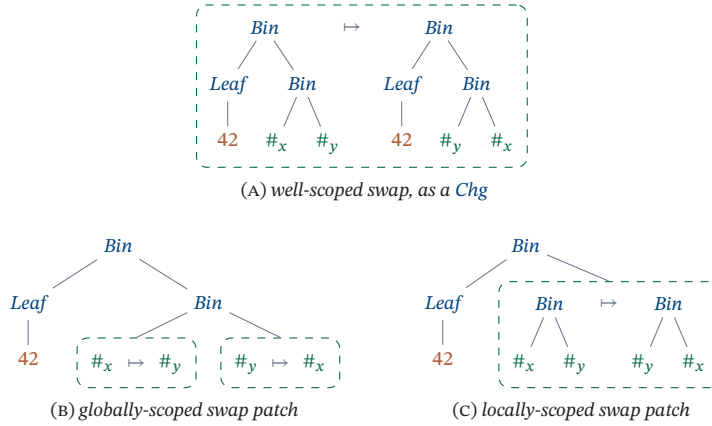


FIGURE 5.13: A change that swaps some elements; naive anti-unification of the deletion and insertion context breaking scoping; and finally the patch with minimal changes.

2576 A *patch*, then, captures the idea of many individual changes operating over separate
 2577 parts of the source tree. It consists in a spine that leads to changes in its leaves, and is
 2578 defined by the type *Patch* below.

2579 **type** *Patch* κ *fam* = *Holes* κ *fam* (*Chg* κ *fam*)

2580 Figure 5.12 illustrates the difference between patches and changes. In Figure 5.12(A)
 2581 we see *Bin* (*Leaf* 42) being repeated in both contexts – whereas in Figure 5.12(B) it has
 2582 been placed in the spine and consequently, is clearly identified as a copy.

2583 Patches are computed from changes by extracting common constructors from the
 2584 deletion and insertion contexts into the spine. In other words, we would like to push
 2585 the changes down towards the leaves of the tree. There are two different ways for doing
 2586 so, illustrated by Figure 5.13. On one hand we can consider the patch metavariables
 2587 to be *globally-scoped*, yielding structurally minimal changes, Figure 5.13(B). On the
 2588 other hand, we could strive for *locally-scoped*, where each change might still contain
 2589 repeated constructors as long as they are necessary to ensure the change is *closed*, as in
 2590 Figure 5.13(C). The first option, *globally-scoped* patches, is very easy to compute. All we
 2591 have to do is to compute the anti-unification of the insertion and deletion context.

2592 $\text{globallyScopedPatch} :: \text{Chg } ki \text{ codes at} \rightarrow \text{Patch}_{PE} \text{ } ki \text{ codes at}$
 $\text{globallyScopedPatch } (\text{Chg } d \ i) = \text{holesMap } (\text{uncurry}' \ \text{Chg}) \ (\text{lge } d \ i)$

2593 *Globally-scoped* patches are easy to compute but contribute little information from
 2594 a synchronization point of view. To an extent, it makes merging even harder. Take

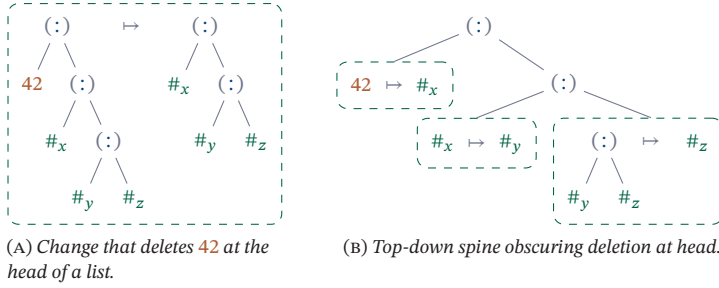


FIGURE 5.14: Globally-scoped patches resulting in misalignment of metavariables due to deletions in the head of linearly-structured data.

Figure 5.14, where a globally scoped patch is produced from a change. It is harder to understand that the $(:42)$ is being deleted by looking at the globally-scoped patch than by looking at the change. This is because the first $(:)$ constructor is considered to be in the spine by the naive anti-unification algorithm, which proceeds top-down. A bottom-up approach is also unpractical, we would have to decide which leaves to pair together and it would suffer similar issues for data inserted on the tail of linearly-structured data.

Locally-scoped patches imply that changes might still contain repeated constructors in the root of their deletion and insertion contexts – hence they will not be structurally minimal. Although more involved to compute, they give us a chance to address insertions and deletions of constructors before we end up misaligning copies.

Independent of global or local scoping, ignoring the information about the spine yields a forgetful functor from patches back into changes, named *chgDistr*. Its definition is straightforward thanks to the free monad structure of *Holes*, which gives us the necessary monadic multiplication. We must take care that *chgDistr* will not capture variables, that is, all metavariables must have already been properly α -converted. We cannot enforce this invariant directly in the *chgDistr* function for performance reasons, consequently, we must manually ensure that all scopes contain disjoint sets of names and therefore can be safely distributed whenever applicable. This is a usual difficulty when handling objects with binders, in general. \triangleleft I wonder how an implementation using De Bruijn indexes would look like. I'm not immediately sure it would be easier to enforce correct indexes. Through the bowels of the code we ensure two changes have disjoint sets of names by adding the successor of the maximum variable of one over the other. \triangleright

```
holesMap :: ( $\forall x . \varphi x \rightarrow \psi x$ )  $\Rightarrow$  Holes  $\kappa$  fam  $\varphi$  at  $\rightarrow$  Holes  $\kappa$  fam  $\psi$  at
holesJoin :: Holes  $\kappa$  fam (Holes  $\kappa$  fam) at  $\rightarrow$  Holes  $\kappa$  fam at
chgDistr :: Patch ki codes at  $\rightarrow$  Chg ki codes at
chgDistr p = Chg (holesJoin (holesMap  $\cdot_{\text{del}}$  p)) (holesJoin (holesMap  $\cdot_{\text{ins}}$  p))
```

The application semantics of *Patch* is independent of the scope choices, and is easily defined in terms of *chgApply*. First we computing a global change that corresponds to the patch in question, then use *chgApply*. The *apply* function below works for locally and globally scoped patches, as long as we care that the precondition for *chgDistr* is maintained.

```

2623  apply :: (All Eq κ) ⇒ Patch κ fam at → SFix κ fam at → Maybe (SFix κ fam at)
2624  apply p = chgApply (chgDistr p)

```

Overall, we find ourselves in a dilemma. On the one hand we have *globally-scoped* patches, which have larger spines but can produce results that are difficult to understand and synchronize, as in Figure 5.14. On the other hand, *locally-scoped* patches are more involved to compute, as we will study next, Section 5.2.1, but they forbid misalignments and also enable us to process small changes independently of one another in the tree. This is particularly important for being able to develop an industrial synchronizer at some point, as it keeps *conflicts* small and isolated.

We propose that the actual solution will consist in using a combination of both local and global scoping. First we will produce a locally-scoped patch, which forbids situations as in Figure 5.14. This patch will consist in an *outer* spine leading to closed locally-scoped changes. This gives us an opportunity to identifying deletions and insertions that could cause copies to be misaligned, essentially producing a globally-scoped *alignment* inside each of those changes. Alignments will be discussed in more detail shortly (Section 5.2.3).

2638 5.2.1 COMPUTING CLOSURES

Computing locally-scoped patches consists of first computing the largest possible spine, like we did with globally-scoped patches, then enlarging the resulting changes until they are well-scoped and closed. Figure 5.13 illustrates this process. Computing the closure of Figure 5.13(A) starts with Figure 5.13(B), then *enlarging* the changes to so that they contain the *Bin* constructor, which fixes their scope (resulting in Figure 5.13(C)).

We say a change is closed when it has no free metavariables and, additionally, its metavariables occur nowhere else. The changes produced by the *chg* function are closed, for example, but they might not be as small as they could be. We say a change is *minimal* when the root constructors in its deletion and insertion context are either different or necessary to maintain scope. Figure 5.15 illustrates different combinations of *closed* and *minimal* changes. The intuition behind *minimal-closed* changes is that two such changes should not interfere with one another.

Producing locally-scoped minimal-closed changes can be difficult under arbitrary renamings. Take Figure 5.15(E), one could argue that: if the occurrences of $\#_x$ in each individual change are, in fact, different, then the changes are minimal-closed. To avoid

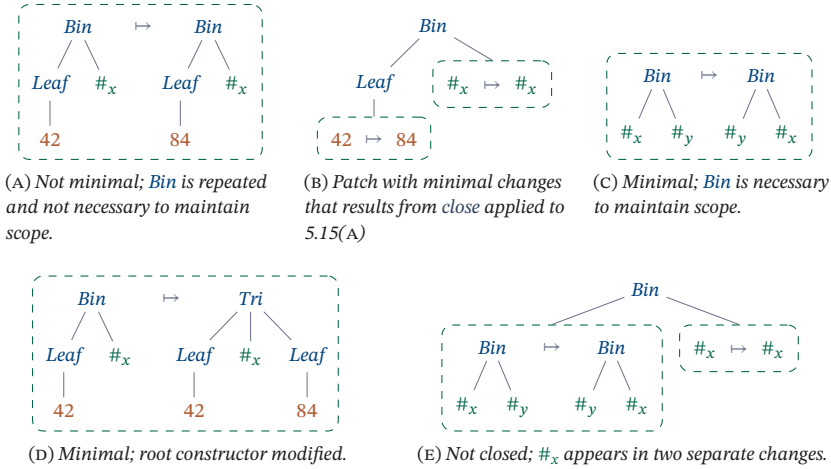


FIGURE 5.15: Some non-minimal-closed and minimal-closed changes examples.

these. In our case, however, we always start from a large well-scoped change produced with *chg*. Consequently, we know that every occurrence of $\#_x$ refers to *the same* tree in the source of the patch. This is another technicality of dealing with names explicitly and provides good reason to enforce that names are always different, even when occurring in separate scopes.

In general, then, we can only know that a change is in fact closed if we know how many times each variable is used globally. Say a variable $\#_z$ is used $n + m$ times in total within a change c , and it has n and m occurrences in the deletion and insertion contexts of c , respectively. Then $\#_z$ does not occur anywhere else but within c , in other words, $\#_z$ is *local* to c . If all variables of c are *local* to c with respect to some global scope, we say c is closed. Given a multiset of variables for the global scope, we can define *isClosedChg* in Haskell as:

```
isClosedChg :: Map Int Arity → Chg κ fam at → Bool
isClosedChg global (Chg d i) = isClosed global (vars d) (vars i)
  where isClosed global ds us = unionWith (+) ds us `isSubmapOf` global
```

The *close* function, shown in Figure 5.16, is responsible for pushing constructors through the least general generalization until they represent minimal-closed changes. It calls an auxiliary version that receives the global scope and keeps track of the variables it has seen so far. The worst case scenario happens when we need *all* constructors of the spine to close the change, in which case, *close* $c = \text{Hole } c$; yet, if we pass a non-well-scoped change to *close*, it cannot produce a result and throws an error instead.

```

close :: Chg  $\kappa$  fam at  $\rightarrow$  Holes  $\kappa$  fam (Chg  $\kappa$  fam) at
close c@(Chg d i) = case closeAux (chgVars c) (holesMap withVars (lgg d i)) of
  InL _  $\rightarrow$  error "invariant failure: c was not well-scoped."
  InR b  $\rightarrow$  holesMap body b

closeAux :: M.Map Int Arity  $\rightarrow$  Holes  $\kappa$  fam (WithVars (Chg  $\kappa$  fam)) at
 $\rightarrow$  Sum (WithVars (Chg  $\kappa$  fam)) (Holes  $\kappa$  fam (WithVars (Chg  $\kappa$  fam))) at
closeAux _ (Prim x) = InR (Prim x)
closeAux gl (Hole cv) = if isClosed gl cv then InR (Hole cv) else InL cv
closeAux gl (Roll x) =
  let aux = repMap (closeAux gl) x
  in case repMapM fromInR aux of
    Just res  $\rightarrow$  InR (Roll res)
    Nothing  $\rightarrow$  let res = chgVarsDistr (Roll (repMap (either' Hole id) aux))
      in if isClosed gl res then InR (Hole res) else InL res
where
  fromInR :: Sum f g x  $\rightarrow$  Maybe (g x)

```

FIGURE 5.16: Complete generic definition of *close* and *closeAux*.

2673 Efficiently computing closures requires us to keep track of the variables that have
 2674 been declared and used in a change – that is, we have seen occurrences in the deletion
 2675 and insertion context respectively. Recomputing this multisets would result in a slower
 2676 algorithm. The *annWithVars* function below computes the variables that occur in two
 2677 contexts and annotates a change with them:

```

data WithVars x at = WithVars {decls , uses :: Map Int Arity , body :: x at}
2678 withVars :: (HolesMV  $\kappa$  fam ::*: HolesMV  $\kappa$  fam) at  $\rightarrow$  WithVars (Chg  $\kappa$  fam) at
withVars (d ::*: i) = WithVars (vars d) (vars i) (Chg d i)

```

2679 The *chgVarsDistr* is the engine of the *close* function. It distributes a spine over a
 2680 change, similar to *chgDistr*, but here we care to maintain the explicit variable annota-
 2681 tions correctly.

```

chgVarsDistr :: Holes  $\kappa$  fam (WithVars (Chg  $\kappa$  fam)) at  $\rightarrow$  WithVars (Chg  $\kappa$  fam) at
chgVarsDistr rs = let us = map (exElim uses) (getHoles rs)
  ds = map (exElim decls) (getHoles rs)
2682 in WithVars (unionsWith (+) ds) (unionsWith (+) us)
  (chgDistr (repMap body rs))

```

2683 The *closeAux* function, Figure 5.16, receives a spine with leaves of type *WithVars* ...
 2684 and attempts to *enlarge* them as necessary. If it is not possible to close the current spine,

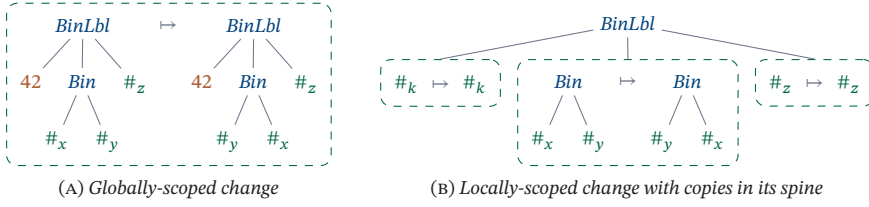


FIGURE 5.17: A Globally-scoped change and the result of applying it to $\text{cpyPrimsOnSpine} \circ \text{close}$, producing a patch with locally scoped changes and copies in its spine.

we return a *InL* ... equivalent to pushing all the constructors of the spine down the deletion and insertion contexts.

5.2.2 THE *diff* FUNCTION

Equipped with the ability to produce changes and minimize them, we move on to defining the *diff* function. As usual, it receives a source and destination trees, *s* and *d*, and computes a patch that encodes the information necessary to transform the source into the destination. The extraction of the contexts yields a *Chg*, which is finally translated to a locally-scoped *Patch* by identifying the largest possible spine, with *close*.

```

diff :: (All Digestible κ) ⇒ SFix κ fam at → SFix κ fam at → Patch κ fam at
diff x y = let dx      = preprocess x
            dy          = preprocess y
            (i, sh)     = buildSharingMap opts dx dy
            (del :*: ins) = extract extMode canShare (dx :*: dy)
            in cpyPrimsOnSpine i (close (Chg del ins))
where canShare t = 1 < treeHeight (getConst (getAnn t))

```

The *cpyPrimsOnSpine* function will issue copies for the opaque values that appear on the spine, as illustrated in Figure 5.17. There, the 42 does not get shared for its height is smaller than 1 but since it occurs in the same location in the deletion and insertion context it can be identified as a copy – which involves issuing a fresh metavariable, hence the parameter *i* in the code above.

5.2.3 ALIGNING CHANGES

As we have seen in the previous sections, locally-scoped changes can avoid misaligning changes (Figure 5.14), but they still do not help us in identifying the insertions and

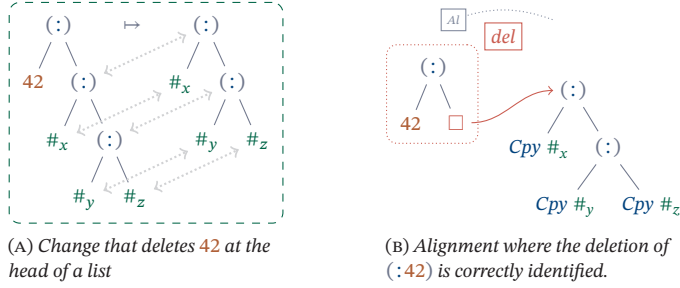


FIGURE 5.18: The change from Figure 5.14, with with an association of which nodes of the deletion and insertion contexts represent the same information, and an explicit representation of that information.

deletions. As it will turn out, identifying these insertions and deletions is crucial for synchronization. In this section we will define a datatype and an algorithm for representing and computing alignments, which make the backbone of synchronization. Untyped synchronizers, such as harmony [32], must employ schemas to identify insertions and deletions avoiding misalignments (Figure 5.14). In our case, the type information enable us to identify insertions and deletions naturally by ensuring that they delete one layer of a *recursive type* at a time, never altering the type of the value under scrutiny.

Take Figure 5.18(A), illustrating the change that motivated locally-scoped patches (Figure 5.14) in the first place. This time, however, arrows connect constructors that represent *the same information* in each respective context. This makes it clear that $(:42)$ has no counterpart in the insertion context and, consequently, corresponds to a deletion. The *Chg* datatype by itself is insufficient to represent all this information. Therefore we need a new datatype for *alignments*, *Al*, and a function that translates a *Chg* into an *Al*. Computing and representing an alignment is, intuitively, the process of computing and representing this association between subtrees of the deletion and insertion contexts. The aligned version of Figure 5.18(A) is shown in Figure 5.18(B), where the *Al* border marks scoping for metavariables. The constructors that are paired up in the deletion and insertion are placed in a spine; those without a correspondent are flagged as deletions or insertions depending on which context they belong. Finally, *Cpy* $\#_\square$ is an abbreviation for *Chg* $\#_\square \#_\square$.

An aligned patch consists of a spine of copied constructors leading to a *well-scoped alignment*. This alignment, in turn, consists of a sequence of insertions, deletions or spines, which finally lead to a *Chg*. These *Chg* in the leaves of the alignment are globally-scoped with respect to the alignment they belong. We also add explicit information about copies and permutations to aid the synchronization engine later. Figure 5.19 illustrates a value of type *Patch* and its corresponding alignment, of type *PatchAl* defined below. Note how the the scope from each change in Figure 5.19(A) is preserved in Figure 5.19(B),

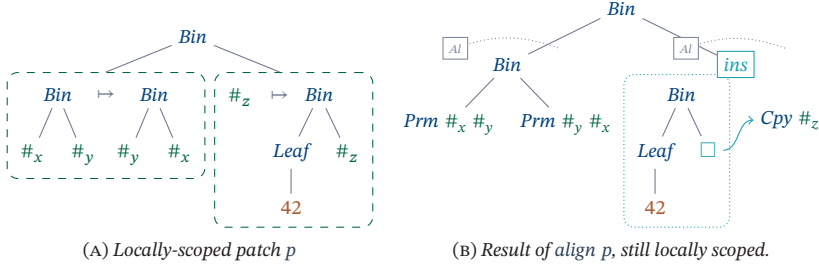


FIGURE 5.19: A patch p and its corresponding aligned version. The Al barrier marks the beginning of an alignment and delimits scopes; copies and permutations are marked explicitly and insertions and deletions indicate their continuation with \square .

but the Bin on the left of the root can now be safely identified as a copy without losing information about the scope of $\#_x$.

type $PatchAl \kappa fam = Holes \kappa fam (Al \kappa fam (Chg \kappa fam))$

Computing the *alignment* for a change c consists in identifying what information in the deletion context correspond to *the same information* in the insertion context. The bits and pieces in the deletion context that have no correspondent in the insertion context should be identified as deletions and vice-versa for the insertion context. In Figure 5.18(A), for example, the second $(:)$ in the deletion context represents the same information as the root $(:)$ in the insertion context.

We can recognize the deletion of $(:42)$ in Figure 5.18(B) structurally. All of its fields, except one recursive field, contains no metavariables. The one subtree which *does contain* metavariables is denoted the *focus* of the deletion (resp. insertion). We denote trees with no metavariables as *rigid* trees. A *rigid* tree has the guarantee that none of its subtrees is being copied, moved or modified. Consequently, *rigid* trees are being entirely deleted from the source or inserted at the destination of the change. If a constructor in the deletion (resp. insertion) context has all but one of its subtrees being *rigid*, it is only natural to consider this constructor to be part of the *deletion* (resp. *insertion*).

Since our patches are locally scoped, computing an aligned patch is simply done by mapping over the spine and aligning the individual changes. Aligning changes, in turn, is done by identifying whether the constructor at the head of the deletion (resp. insertion) context can be deleted (resp. inserted) then recursing on the focus of the deletion (resp. insertion). When the root of the deletion context and the root of the insertion context qualify for deletion and insertion, we check whether we can add them to a spine instead.

2753 5.2.3.1 GENERIC ALIGNMENTS

2754 We will be representing a deletion or insertion of a recursive *layer* by identifying the
 2755 *position* where this modification must take place. Moreover, said position must be a
 2756 recursive field of the constructor – that is, the deletion or insertion must not alter the type
 2757 that our patch operates over. This is easy to identify since we followed typed approach,
 2758 where we always have access to type-level information.

2759 In the remainder of this section we discuss the datatypes necessary to represent an
 2760 aligned change, as illustrated in Figure 5.18(B), and how to compute said alignments
 2761 from a *Chg* κ *fam* *at*. The *alignChg* function, declared below, will receive a well-scoped
 2762 change and compute an alignment.

2763 $\text{alignChg} :: \text{Chg } \kappa \text{ fam } at \rightarrow \text{Al } \kappa \text{ fam } (\text{Chg } \kappa \text{ fam}) at$

2764 The alignments here, encoded in the *Al* datatype, is similar to its predecessor *Al μ*
 2765 from *stdiff* (Section 4.1.2), it records insertions, deletions and spines over a fixpoint.
 2766 Insertions and deletions will be represented with *Zippers* [41]. A zipper over a datatype
 2767 *t* is the type of *one-hole-contexts* over *t*, where the hole indicates a focused position.
 2768 We will use the zippers provided directly by the *generics-simplistic* library (Sec-
 2769 tion 3.2.4.1). These zippers encode a *single* layer of a fixpoint at a time, for example, a
 2770 zipper over the *Bin* constructor is either *Bin* \square *u* or *Bin* *u* \square , indicating the focus is in
 2771 either the left or the right subtree. It *does not* enable us specify a nested focus point, like
 2772 in *Bin* (*Bin* \square *t*) *u*.

2773 A value of type *Zipper* *c* *g* *h* *at* is then equivalent to a constructor of type *at* with
 2774 one of its recursive positions replaced by a value of type *h* *at* and the other positions *at'*
 2775 (recursive or not) carrying values of type *g* *at'*. The *c* above is a constraint that enables
 2776 us to inform GHC some properties of type *at* and is mostly a technicality.

2777 An alignment *Al* κ *fam* *f* *at* represents a sequence of insertions and deletions inter-
 2778 leaved with spines, copies and permutations which ultimately lead to *unclassified modi-*
 2779 *fications*, which are typed according to the *f* parameter. Next, we will go through the six
 2780 constructors of *Al* one by one. First we have deletions and insertions, which explicitly
 2781 mention a zipper and one recursive field to continue the alignment.

2782 $\text{data Al } \kappa \text{ fam } f \text{ at where}$
 $\text{Del} :: \text{Zipper } (\text{CompoundCnstr } \kappa \text{ fam } at) (\text{SFix } \kappa \text{ fam}) (\text{Al } \kappa \text{ fam } f) at \rightarrow \text{Al } \kappa \text{ fam } f at$
 $\text{Ins} :: \text{Zipper } (\text{CompoundCnstr } \kappa \text{ fam } at) (\text{SFix } \kappa \text{ fam}) (\text{Al } \kappa \text{ fam } f) at \rightarrow \text{Al } \kappa \text{ fam } f at$

2783 The *CompoundCnstr* constraint above must be carried around to indicate we are
 2784 aligning a type that belongs to the mutually recursive family and therefore has a generic
 2785 representation – again, just a Haskell technicality.

2786 Naturally, if no insertion or deletion can be performed but both insertion and dele-
 2787 tion contexts have the same constructor at their root, we want to recognize this construc-
 2788 tor as part of the spine of the alignment, and continue to align its fields pairwise.

2789 $Spn :: (CompoundCnstr \kappa fam x) \Rightarrow SRep (Al \kappa fam f) (Rep at) \rightarrow Al \kappa fam f at$

2790 The *Spn* inside an alignment does not need to preserve metavariable scoping, conse-
 2791 quently, it can be pushed closer to the leaves uncovering as many copies as possible.

2792 When no *Ins*, *Del* or *Spn* can be used, we must be fallback to recording a unclassified
 2793 modification, of type *f at*. Most of the times *f* will be simply *Chg* κfam , but we will be
 2794 needing to add some extra information in the leaves of an alignment later. Moreover,
 2795 keeping the *f* a parameter turns *Al* into a functor which enables us to map over it easily.

2796 $Mod :: f at \rightarrow Al \kappa fam f at$

2797 Imagine an alignment $a = Mod (Chg \#_x \#_y)$. Does *a* represent a copy or is *x*
 2798 contracted or duplicated? Because metavariables are scoped globally within an align-
 2799 ment, we can only distinguish between copies and duplications by traversing the entire
 2800 alignment and recording the arity of *x*. Yet, it is an important distinction to make. A
 2801 copy synchronizes with anything whereas a contraction needs to satisfy additional con-
 2802 straints. Therefore, we will identify copies and permutations directly in the alignment
 2803 to aid the merge function, later.

2804 Let $c = Chg \#_x \#_y$ with both *x* and *y* occur twice in their global scope: once in the
 2805 deletion context and once in the insertion context. We say *c* is a copy when $x \equiv y$ and a
 2806 permutation when $x \not\equiv y$. These are the last two constructors of *Al*.

2807 $Cpy :: Metavar \kappa fam at \rightarrow Al \kappa fam f at$
 $Prm :: Metavar \kappa fam at \rightarrow Metavar \kappa fam at \rightarrow Al \kappa fam f at$

2808 Equipped with a definition for alignments, we move on to defining *alignChg*. Given
 2809 a change *c*, the first step of *alignChg c* is checking whether the root of c_{del} (resp. c_{ins}) can
 2810 be deleted (resp. inserted). A deletion (resp. insertion) of an occurrence of a constructor
 2811 *X* can be performed when all the of fields of *X* at this occurrence are *rigid* trees with the
 2812 exception of a single recursive field – recall *rigid* trees contains no metavariables. If we
 2813 can delete the root, we flag it as a deletion and continue through the recursive *non-rigid*
 2814 field. If we cannot perform a deletion at the root of c_{del} nor an insertion at the root of
 2815 c_{ins} but they are constructed with the same constructor, we identify the constructor as
 2816 being part of the alignments' spine. If c_{del} and c_{ins} do not even have the same constructor
 2817 at the root, nor are copies or permutations, we finally fallback and flag an unclassified
 2818 modification.

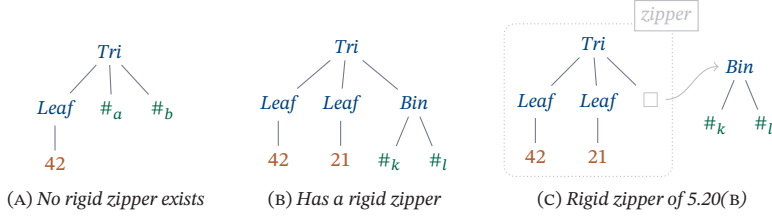


FIGURE 5.20: Example calls to `hasRigidZipper` and their respective return values where applicable.

2819 To check whether constructors can be deleted or inserted efficiently, we must anno-
 2820 tate rigidity information throughout our trees. The *IsRigid* datatype captures whether
 2821 a tree contains any metavariables or not and is placed in every node of a tree with the
 2822 *annotRigidity* function.

2823 **type** *IsRigid* = *Const Bool*
annotRigidity :: *Holes* κ *fam* *h* *x* \rightarrow *HolesAnn* κ *fam* *IsRigid* *h* *x*

2824 After annotations the trees with rigidity information, we extract the zippers that
 2825 witness potential insertions or deletions. This is done by the *hasRigidZipper* function,
 2826 which is implemented by extracting *all* possible zippers from the root and checking
 2827 whether there is one such that all of its fields are rigid except for the focus of the zip-
 2828 per. If we find such a zipper, we return it wrapped in a *Just*. When a rigid zipper exists
 2829 it is unique by definition, hence there is no choice involved in detecting insertions and
 2830 deletions, which keeps our algorithms efficient and deterministic.

2831 Figure 5.20 exemplifies two possible arguments to *hasRigidZipper*. The tree in Fig-
 2832 ure 5.20(A) has three possible zippers: focusing on either of its recursive positions. Nei-
 2833 ther of them, however, would have all its subtrees rigid except the focus point. Fig-
 2834 ure 5.20(B) on the other hand has one of its zippers (the one with focus on *Bin* $\#_k$ $\#_l$, Fig-
 2835 ure 5.20(C)) rigid, that is, none of the trees within the zipper has any metavariables. We
 2836 omit the full implementation of *hasRigidZipper* but invite the interested reader should
 2837 check *Data.HDiff.Diff.Align* in the source code (Appendix A).

2838 Checking for deletions, then, can be easily done by first checking whether the root
 2839 can has a rigid zipper, if so, we can flag the deletion. In the excerpt of *alD* below, if *d*
 2840 was the tree in Figure 5.20(B), *focus* would be *Bin* $\#_k$ $\#_l$, which is the single *non-rigid*
 2841 recursive subtree of *d*.

2842 *alD* *d* *i* = **case** *hasRigidZipper* *d* **of**
Just (*Zipper* *zd* *focus*) \rightarrow *Del* *zd* (*continueAligning* *focus* *i*)

2843 The complete *alD* is more involved. For one, we must check whether *i* also has a
 2844 rigid zipper and when both *d* and *i* have rigid zippers, we must check whether they are
 2845 the same constructor and, if so, mark it as part of the spine instead. The *al* function en-
 2846 capsulates the *alD* above and is shown in Figure 5.21. A call to *al* will attempt to extract
 2847 deletions, then insertions, then finally falling back to copies, permutations, modifica-
 2848 tions or recursively calling itself inside spines.

2849 To compute an alignment, then, we start computing the multiset of variables used
 2850 throughout a patch, annotate the deletion and insertion context with *IsRigid* and pass
 2851 everything to the *al* function.

```
2852 alignChg :: Chg κ fam at → Al κ fam (Chg κ fam) at
alignChg c@(Chg d i) = al (chgVargs c) (annotRigidity d) (annotRigidity i)
```

2853 Forgetting information computed *alignChg* is trivial but enables us to convert back
 2854 into a *Chg*. The *disalign* function, sketched below, plugs deletion and insertion zippers
 2855 casting a zipper over *SFix* into a zipper over *Holes* where necessary; distributes the con-
 2856 structors in the spine into both deletion and insertion contexts and translates *Cpy* and
 2857 *Prm* as expected.

```
disalign :: Al κ fam (Chg κ fam) at → Chg κ fam at
disalign (Del (Zipper del rest)) =
2858   let Chg d i = disalign rest
      in Chg (Roll (plug (cast del) d) i)
disalign ...
```

2859 Distributing an outer spine through an alignment is trivial. All we must do is place
 2860 all the constructors of the outer spine as *Spn*:

```
alDistr :: PatchAl κ fam at → Al κ fam (Chg κ fam) at
alDistr (Hole al) = al
2861 alDistr (Prim k) = Spn (Prim k)
alDistr (Roll r) = Spn (Roll (repMap alDistr r))
```

2862 Finally, computing aligned patches from locally-scoped patches is done by mapping
 2863 over the outer spine and aligning the changes individually, then we make a pass over the
 2864 result and issue copies for opaque values that appear on the alignment's inner spine.

```
align :: Patch κ fam at → PatchAl κ fam at
2865 align = fst ∘ align'
```

2866 The auxiliary function *align'* returns the successor of the last issued name to en-
 2867 sure we can easily produce fresh names later on, if need be. Once again, a technical-

```

type Aligner  $\kappa$  fam = HolesAnn  $\kappa$  fam IsStiff (Metavar  $\kappa$  fam) t
    → HolesAnn  $\kappa$  fam IsStiff (Metavar  $\kappa$  fam) t
    → Al  $\kappa$  fam (Chg  $\kappa$  fam t)

al :: Map Int Arity → Aligner  $\kappa$  fam
al vars d i = alD (alS vars (al vars)) d i
where
    -- Try deleting many; try inserting one; decide whether to delete,
    -- insert or spn in case both Del and Ins are possible. Fallback to
    -- inserting many.
    alD :: Aligner  $\kappa$  fam → Aligner  $\kappa$  fam
    alD f d i = case hasRigidZipper d of -- Is the root a potential deletion?
        Nothing      → al f d i
        -- If so, we must check whether we also have a potential insertion.
        Just (Zipper zd rd) → case hasRigitZipper i of
            Nothing      → Del (Zipper zd (alD f rd i))
            Just (Zipper zi ri) → case zipSZip zd zi of -- are zd and zi the same?
                Just res → Spn $ plug (zipperMap Mod res) (alD f rd ri)
                Nothing → Del (Zipper zd (Ins (Zipper zi (alD f rd ri))))

    -- Try inserting many; fallback to parametrized action.
    alI :: Aligner  $\kappa$  fam → Aligner  $\kappa$  fam
    alI f d i = case hasRigidZipper i of
        Nothing      → f d i
        Just (Zipper zi ri) → Ins (Zipper zi (alI f d ri))

    -- Try extracting spine and executing desired action
    -- on the leaves; fallback to deleting; inserting then modifying
    -- if no spine is possible.
    alS :: Map Int Arity → Aligner  $\kappa$  fam → Aligned  $\kappa$  fam
    alS vars f d@(Roll' _ sd) i@(Roll' _ si) =
        case zipSRep sd si of
            Nothing → alMod vars d i
            Just r  → Spn (repMap (uncurry' f) r)
    syncSpine vars _ d i = alMod vars d i

    -- Records a modification, copy or permutation.
    alMod :: Map Int Arity → Aligned  $\kappa$  fam
    alMod vars (Hole' _ vd) (Hole' _ vi) =
        -- are both vd and vi with arity 2?
        | all (≡ Just 2 ∘ flip lookup vars) [metavarGet vd, metavarGet vi]
        → if vd ≡ vi then Cpy vd else Prm vd vi
        | otherwise
        → Mod (Chg (Hole vd) (Hole vi))
    alMod _ _ d i = Mod (Chg d i)

```

FIGURE 5.21: Complete definition of *al*.

ity of handling names explicitly. Note that *align* introduces information, namely, new metavariables that represent copies over opaque values that appear on the alignment's spine. This means that mapping *disalign* to the result of *align* will *not* produce the same result. Alignments and changes are *not* isomorphic.

```

align' :: Patch κ fam at → (PatchAl κ fam at , Int)
align' p = flip runState maxv $ holesMapM (alRefineM cpyPrims ∘ alignChg vars) p
where vars = patchVars p
      maxv = maybe 0 id (lookupMax vars)

```

The *cpyPrims* above issues a *Cpy* *i*, for a fresh name *i* whenever it sees a modification with the form *Chg* (*Prim* *x*) (*Prim* *y*) with *x* ≡ *y*. The *alRefineM* *f* applies a function in the leaves of the *Al* and has type.

```

alRefineM :: (Monad m) ⇒ (∀ x . f x → m (Al κ fam g x))
           → Al κ fam f ty → m (Al κ fam g ty)

```

This process of computing alignments showcases an important aspect of our well-typed approach: the ability to access type-level information in order to compute zippers and understand deletions and insertions of a single layer in a homogeneous fashion – the type that results from the insertion or deletion is the same type that is expected by the insertion or deletion.

5.2.4 SUMMARY

In Section 5.2 we have seen how *Chg* represents an unrestricted tree-matching, which can later be translated into isolated, well-scoped, fragments connected through an outer spine and making up a *Patch*. Finally, we have seen how to extract valuable information from well-scoped about which constructors have been deleted, inserted or still belong to an inner spine, giving rise to alignments. This representation is a mix of local and global alignments. The outer spine is important to isolate a large change into smaller chunks, independent of one another.

The *diff* function produces a *Patch* instead of a *PatchAl* to keep it consistent with our previously published work [71], but also because its easier to manage calls to *align* where they are directly necessary, since *align* produces fresh variables and this can require special attention to keep names from being shadowed.

In fact, the *diff* function could be any path in the diagram portrayed in Figure 5.22. There is no *right* choice as this depends on the specific application in question. For our particular case of pursuing a synchronization function, we require all the information up to *PatchAl*.

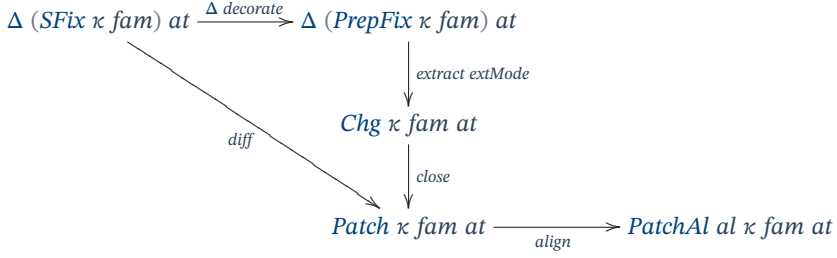


FIGURE 5.22: Conceptual pipeline of the design space for the *diff* function. $\Delta f x$ denotes $(f x, f x)$

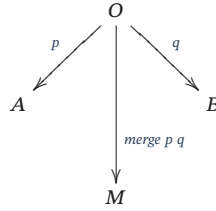


FIGURE 5.23: Span of patches, p (transforms O into A) and q (transforms O into B). Both patches have a common element O in their domain. The patch merge $p q$ applies to this common ancestor O and can be thought of as the union of the changes of p and q .

5.3 MERGING ALIGNED PATCHES

In this section we will be exploring a synchronization algorithm for aligned patches, witnessed by the *merge* function, declared below, which receives two *aligned* patches p and q that make a span – that is, have at least one common element in their domain. The result of *merge p q* is a patch that can might contain conflicts, denoted by *PatchC*, whenever both p and q modify the same subtree in two distinct ways. If p and q do not make a span *merge p q* returns *Nothing*. Figure 5.23 illustrates a span of patches p and q and their merge which is supposed to be applied to their common ancestor producing a tree which combines the modifications performed by p and q , when possible.

merge :: *PatchAl* κ fam at \rightarrow *PatchAl* κ fam al \rightarrow *Maybe* (*PatchC* κ fam at)

Recall our patches consist of a spine which leads to locally-scoped alignments, which in turn have an inner spine that ultimately leads to changes. The distinction between

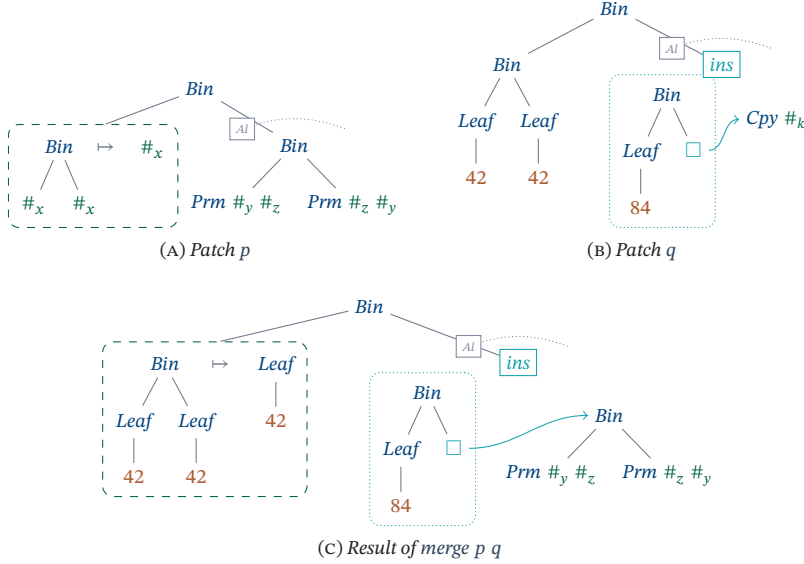


FIGURE 5.24: Example of a simple synchronization

the *outer* spine and the spine inside the alignments is the scope. Consequently, we can map a pure function over the outer spine without having to carry information about local scopes to the next call. When manipulating the *inner* spine, however, we must keep track of which variables have or have not been declared or used. Take the example in Figure 5.24, that merges patches p (Figure 5.24(A)) and q (Figure 5.24(B)) to produce a new patch (Figure 5.24(C)). While synchronizing the left child of each root, we discover that the tree located at (or, identified by) $\#_x$ was *Leaf* 42. We must remember this information since we will encounter $\#_x$ again and must ensure that it matches with its previously discovered value in order to perform the contraction. When we finish synchronizing the left child of the root, though, we can forget about $\#_x$ since well-scopedness of alignments guarantees $\#_x$ will not appear elsewhere.

It helps to think about metavariables in a change as a unique identifier for a subtree in the source. For example, if one change modifies a subtree x into a different subtree x' , but some other change moves x , identified by $\#_x$, to a different location in the tree, the result of synchronizing these should be the transport of x' into the new location – which is exactly where $\#_x$ appears in the insertion context. The example in Figure 5.25 illustrates this very situation: the source tree identified by $\#_x$ in the deletion context of Figure 5.25(B) was changed, by Figure 5.25(A), from *Leaf* 42 into *Leaf* 84. Since p altered the content of a subtree, but q altered its location, they are *disjoint* – they alter different

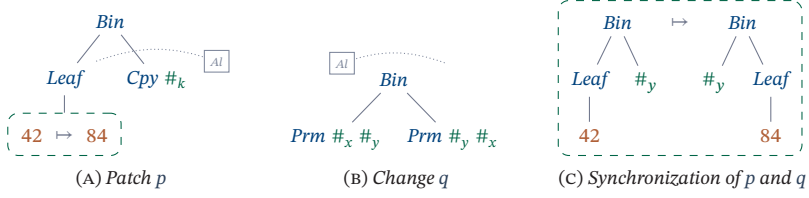


FIGURE 5.25: Example of a simple synchronization.

2929 aspects of the common ancestor. Hence, the synchronization is possible and results in
 2930 Figure 5.25(C).

2931 Given then two aligned patches, the *merge* $p\ q$ function below will map over the
 2932 common prefix of the spines of p and q , captured by their least-general-generalization
 2933 and produce a patch with might contain conflicts inside. \triangleleft In the actual implementation
 2934 we receive two patches and align them inside merge, this helps ensuring they will have a
 2935 disjoint set of names. \triangleright

```

merge :: PatchAl  $\kappa$  fam at  $\rightarrow$  PatchAl  $\kappa$  fam at  $\rightarrow$  Maybe (PatchC  $\kappa$  fam at)
merge oa ob = holesMapM (uncurry' go) (lgg oa ob)
2936   where go :: Holes  $\kappa$  fam (Al  $\kappa$  fam) at  $\rightarrow$  Holes  $\kappa$  fam (Al  $\kappa$  fam) at
            $\rightarrow$  Maybe (Sum (Conflict  $\kappa$  fam) (Chg  $\kappa$  fam) at)
           go ca cb = mergeAl (alDistr ca) (alDistr cb)
```

2937 A conflict, defined below, contains a label identifying which branch of the merge al-
 2938 gorithm issued it and the two alignments that could not be synchronized. Conflicts are
 2939 issued whenever we were not able to reconcile the alignments in question. This happens
 2940 either when we cannot detect that two edits to the same location are non-interfering or
 2941 when two edits to the same location in fact interfere with one another. Putting it differ-
 2942 ently, conflicts might contain false positives where edits could have been automatically
 2943 reconciled. The *PatchC* datatype encodes patches which might contain conflicts inside.

```

2944   data Conflict  $\kappa$  fam at = Conflict String (Al  $\kappa$  fam at) (Al  $\kappa$  fam at)
   type PatchC  $\kappa$  fam at = Holes  $\kappa$  fam (Sum (Conflict  $\kappa$  fam) (Chg  $\kappa$  fam)) at
```

2945 Merging has a large design space. In what follows we will discuss our initial explo-
 2946 ration and prototype algorithm, which was driven practical experiments (Chapter 6). \triangleleft
 2947 Unfortunately, I never had time to come back and refine the merging algorithm from its
 2948 prototype phase into a more polished version. The merging algorithm was the last aspect
 2949 of the project I worked on. \triangleright

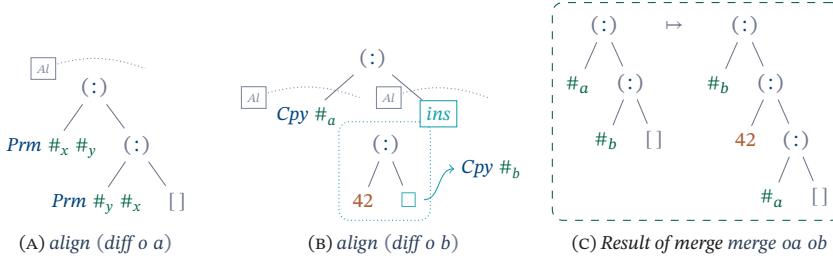


FIGURE 5.26: Example merge of two simple patches.

The *mergeAl* function is responsible for synchronizing alignments and is where most of the work is happens. In broad strokes, it is similar to synchronizing *Patch_{st}*'s, Section 4.2: insertions are preserved as long as they do not happen simultaneously. Deletions must be *applied* before continuing and copies are the identity of synchronization. In the current setting, however, we also have permutations and arbitrary changes to look at. The general conducting line of our synchronization algorithm is to first record how each subtree was modified and then instantiate these modifications in a later phase. Traversing the patches simultaneously whilst constructing substitutions would not suffice since the order which metavariables appear in each context can be drastically different. This would require us to start over every time we discovered new information on the current traversal, yielding a very slow merging algorithm.

Let us look at an example, illustrated in Figure 5.26. We start identifying we are in a situation where both *diff o a* and *diff o b* are spines, that is, they copy the same constructor at their root. Recursing pairwise through their children, we see a permutation versus a copy, since a copy is the identity element, we return the permutation. On the right we see another spine versus an insertion, but since the insertion represents new information, it must be preserved. Finally, inside the insertion we see another copy, which means that the spine should be preserved as is. The resulting patch can be seen in Figure 5.26(C).

We keep track of the equivalences we discover in a state monad. The instantiation of metavariables will be stored under *inst* and the list of tree equivalences will be stored under *eqs*.

```

data MergeState κ fam = MergeState
  { inst :: Map (Exists (Metavar κ fam)) (Exists (Chg      κ fam))
  , eqs  :: Map (Exists (Metavar κ fam)) (Exists (HolesMV κ fam))
  }

```

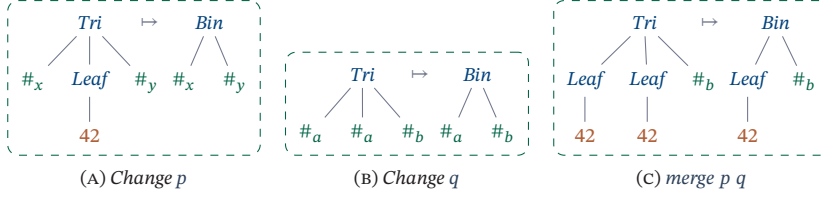


FIGURE 5.27: Merging arbitrary changes requires knowledge of equivalences between metavariables and trees.

It is important to keep track of equivalences in *eqs*. Say, for example, we are to merge two changes that were left as *unclassified* by our alignment algorithm. Naturally, their deletion contexts must be unifiable, yielding a series of equivalences between their metavariables but since we do not possess information about exactly how each of those metavariables were transformed, we cannot register how they changed in *inst*. Figure 5.27 provides a simple such example. When unifying the deletion contexts of Figure 5.27(A) and Figure 5.27(B), we learn that $\{\#_x \equiv \text{Leaf } 42, \#_a \equiv \#_x; \#_b \equiv \#_y\}$, which enable us to conclude both changes are compatible and perform the same action modulo a contraction and can be merged, yielding Figure 5.27(C)

Conflicts and errors stemming from the arguments to *mergeAl* not forming a span will be distinguished by the *MergeErr* datatype, below. We also define auxiliary functions to raise each specific error in a computation inside the *Except* monad.

```
data MergeErr = NotASpan | Conf String
```

```
throwConf lbl = throwError (Conf lbl)
throwNotASpan = throwError NotASpan
```

The *mergeAl* function is defined as a wrapper around *mergeAlM*, which is defined in terms of the *MergeM* monad to help carry around the necessary state and raises errors through the *Except* monad.

```
type MergeM κ fam = StateT (MergeState κ fam) (Except MergeErr)
mergeAl :: Aligned κ fam x → Aligned κ fam x
      → Maybe (Sum (Conflict κ fam) (Chg κ fam) x)
mergeAl x y = case runExcept (evalStateT (mergeAlM p q) mrgStEmpty) of
  Left NotASpan → Nothing
  Left (Conf err) → Just (InL (Conflict err p q))
  Right r → Just (InR (disalign r))
```

Finally, the *mergeAlM* function maps over both alignments that we wish to merge and collects all the constraints and observations. It then attempts to splits these con-

2992 straints and observations into two maps: (A) a deletion map that contains information
 2993 about what a subtree identified by a metavariable *was*; and (B) an insertion map that
 2994 identifies what said metavariable *became*. If it is possible to produce these two idempotent
 2995 substitutions, it then makes a second pass computing the final result.

```

mergeAlM :: Al κ fam at → Al κ fam at → MergeM κ fam (Al κ fam at)
mergeAlM p q = do phase1 ← mergePhase1 p q
                info ← get
2996                case splitDelInsMap info of
                    Left _ → throwConf "failed-contr"
                    Right di → alignedMapM (mergePhase2 di) phase1

```

2997 **FIRST PHASE.** The first pass is computed by the *mergePhase1* function, which will pop-
 2998 ulate the state with instantiations and equivalences and place values of type *Phase2* in-
 2999 place in the alignment. These values instruct the second phase on how to proceed on
 3000 that particular location. Before proceeding, though, we must process the information
 3001 we gathered into a deletion and an insertion map, with *splitDelInsMap* function. First
 3002 we look into how the first pass instantiates metavariables and registers equivalences.

3003 The *mergePhase1* function receives two alignments and produces a third alignment
 3004 with instructions for the *second phase*. These instructions can be instantiating a change,
 3005 with *P2Instantiate*, which might include a context to ensure for some consistency predi-
 3006 cates. Or checking that two changes are α -equivalent after they have been instantiated.

```

data Phase2 κ fam at where
3007   P2Instantiate :: Chg κ fam at → Maybe (HolesMV κ fam at) → Phase2 κ fam at
   P2TestEq      :: Chg κ fam at → Chg κ fam at → Phase2 κ fam at

```

3008 Deciding which instruction should be performed depends on the structure of the
 3009 alignments under synchronization, and is done by the *mergePhase1* function, whose
 3010 cases will be discussed one by one, next.

```

mergePhase1 :: Al κ fam x → Al κ fam x → MergeM κ fam (Al' κ fam (Phase2 κ fam) x)
mergePhase1 p q = case (p, q) of
3011   (Cpy _, _) → return (Mod (P2Instantiate (disalign q)))
   (_, Cpy _) → return (Mod (P2Instantiate (disalign p)))

```

3012 The first cases we have to handle are copies, shown above, which should be the iden-
 3013 tity of synchronization. That is, if *p* is a copy, all we need to do is modify the tree accord-
 3014 ing to *q* at the current location. We might need to refine *q* according to other constraints
 3015 we discovered in other parts of the alignment in question, so the final instruction is to

3016 *instantiate* the *Chg* that comes from forgetting the alignment *q*. Recall *disalign* maps
 3017 alignments back into changes.

3018 Next we look at permutations, which are almost copies in the sense that they do not
 3019 modify the *content* of the tree, but they modify the *location*. We distinguish the case
 3020 where both patches permute the same tree versus the case where one patch permutes
 3021 the tree but the other changes its contents.

3022
$$\begin{aligned} (\text{Prm } x \ y, \text{Prm } w \ z) &\rightarrow \text{Mod } \langle \$ \rangle \text{ mrgPrmPrm } x \ y \ w \ z \\ (\text{Prm } x \ y, _) &\rightarrow \text{Mod } \langle \$ \rangle \text{ mrgPrm } x \ y \ (\text{disalign } q) \\ (_, \text{Prm } x \ y) &\rightarrow \text{Mod } \langle \$ \rangle \text{ mrgPrm } x \ y \ (\text{disalign } p) \end{aligned}$$

3023 If we are to merge two permutations, *Prm* $\#_x \#_y$ against *Prm* $\#_w \#_z$, for example, we
 3024 know that $\#_x$ and $\#_w$ must refer to the same subtree, hence we register their equivalence.
 3025 But since the two changes permuted the same tree, we can only synchronize them if they
 3026 were permuted to the *same place*, in other words, if both permutations turn out to be
 3027 equal at the end of the synchronization process. Consequently, we issue a *P2TestEq*.

3028
$$\begin{aligned} \text{mrgPrmPrm} &:: \text{Metavar } \kappa \text{ fam } x \rightarrow \text{Metavar } \kappa \text{ fam } x \\ &\rightarrow \text{Metavar } \kappa \text{ fam } x \rightarrow \text{Metavar } \kappa \text{ fam } x \\ &\rightarrow \text{MergeM } \kappa \text{ fam } (\text{Phase2 } \kappa \text{ fam } x) \\ \text{mrgPrmPrm } x \ y \ w \ z &= \text{onEqvs } (\lambda \text{eqs} \rightarrow \text{substInsert eqs } x \ (\text{Hole } w)) \\ &\quad > \text{return } (\text{P2TestEq } (\text{Chg } (\text{Hole } x) \ (\text{Hole } y)) \ (\text{Chg } (\text{Hole } w) \ (\text{Hole } z))) \end{aligned}$$

3029 If we are merging one permutation with something other than a permutation, how-
 3030 ever, we know one change modified the location of a tree, whereas another potentially
 3031 modified its contents. All we must do is record that the tree identified by $\#_x$ was modified
 3032 according to *c*. After we have made one entire pass over the alignments being merged,
 3033 we must instantiate the permutation with the information we discovered – the $\#_x$ oc-
 3034 currence in the deletion context of the permutation will be c_{del} , potentially simplified or
 3035 refined. The $\#_y$ appearing in the insertion context of the permutation will be instanti-
 3036 ated with whatever we come to discover about it later. We know there *must* be a single
 3037 occurrence of $\#_y$ in a deletion context because the alignment flagged it as a permutation.

3038
$$\begin{aligned} \text{mrgPrm} &:: \text{Metavar } \kappa \text{ fam } x \rightarrow \text{Metavar } \kappa \text{ fam } x \rightarrow \text{Chg } \kappa \text{ fam } x \\ &\rightarrow \text{MergeM } \kappa \text{ fam } (\text{Phase2 } \kappa \text{ fam } x) \\ \text{mrgPrm } x \ y \ c &= \text{addToInst } \text{"prm-chg"} \ x \ c \\ &\quad > \text{return } (\text{P2Instantiate } (\text{Chg } (\text{Hole } x) \ (\text{Hole } y)) \ \text{Nothing}) \end{aligned}$$

3039 The *addToInst* function inserts the (x, c) entry in *inst* if *x* is not yet a member. It
 3040 raises a conflict with the supplied label if *x* is already in *inst* with a value that is different
 3041 than *c*. \triangleleft *I believe that we could develop a better algorithm if instead of forbidding values*
 3042 *different than c we check to see whether the two different values can also be merged. I*

3043 *ran into many difficulties tracking how subtrees were moved and opted for the easy and*
 3044 *pragmatic option of not doing anything difficult here. ▷*

3045 The call to *addToInst* in *mrgPrm* never raises a “*prm-chg*” conflict. This is because
 3046 $\#_x$ and $\#_y$ are classified as a permutation – each variable occurs exactly once in the
 3047 deletion and once in the insertion contexts. Therefore, it is impossible that x was already
 3048 a member of *inst*. ◁ *In fact, throughout our experiments, in Chapter 6, we observed that*
 3049 *“prm-chg” never showed up as a conflict in our whole dataset, as expected. ▷*

3050 With permutations and copies out of the way, we start looking at the more intricate
 3051 branches of the merge function. Insertions are still fairly simple and must preserved as
 3052 long as they do not attempt to insert different information in the same location – we
 3053 would not be able to decide which insertion come first in this situation.

3054
$$\begin{aligned} & (Ins (Zipper\ z\ p'), Ins (Zipper\ z'\ q')) \\ & \quad | z \equiv z' \quad \rightarrow Ins \circ Zipper\ z \langle \$ \rangle mergePhase1\ p'\ q' \\ & \quad | otherwise \quad \rightarrow throwConf\ \textcolor{brown}{“ins-ins”} \\ & (Ins (Zipper\ z\ p'), -) \rightarrow Ins \circ Zipper\ z \langle \$ \rangle mrgPhase1\ p'\ q \\ & (-, Ins (Zipper\ z\ q')) \rightarrow Ins \circ Zipper\ z \langle \$ \rangle mrgPhase1\ p\ q' \end{aligned}$$

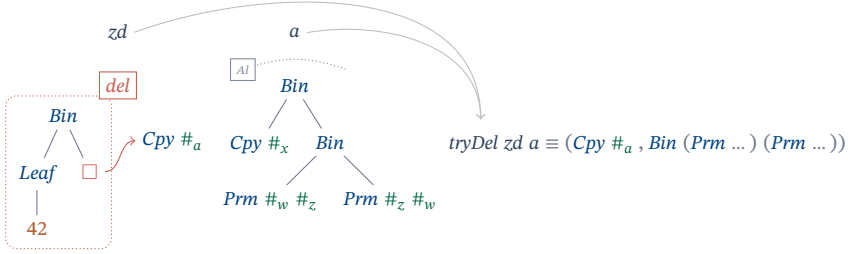
3055 Deletions must be preserved and *executed*. That is, if one patch deletes a constructor
 3056 but the other modifies the fields the constructor, we must first ensure that none of the
 3057 deleted fields have been modified but the deletion should be preserved in the merge.
 3058 The *tryDel* function attempts to execute the deletion of a zipper over an alignment, and,
 3059 if successful, returns the pair of alignments we should continue to merge. It essentially
 3060 overlaps the deletion zipper with a and observe whether a performs no modifications
 3061 anywhere except on the focus of the zipper. When its not possible to execute the deletion
 3062 we can continue. Figure 5.28 illustrate some example calls to *tryDel*, whose complete
 3063 generic definition is shown in Figure 5.29.

3064
$$\begin{aligned} & (Del\ zp@(Zipper\ z\ -), -) \rightarrow Del \circ Zipper\ z \langle \$ \rangle (tryDel\ zp\ q \succcurlyeq uncurry\ mrgPhase1) \\ & (-, Del\ zq@(Zipper\ z\ -)) \rightarrow Del \circ Zipper\ z \langle \$ \rangle (tryDel\ zq\ p \succcurlyeq uncurry\ mrgPhase1) \end{aligned}$$

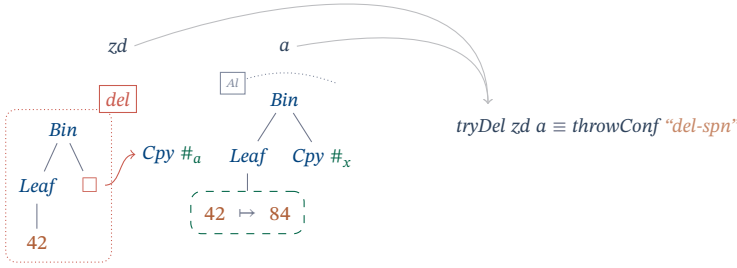
3065 Note that since *merge* is symmetric, we an freely swap the order of arguments. ◁ *Let*
 3066 *me rephrase that. The merge should be symmetric, and QuickCheck tests were positive*
 3067 *of this, but I have not come to the point of proving this yet. ▷*

3068 Next we have spines versus modifications. Intuitively, we want to match the dele-
 3069 tion context of the change against the spine and, when successful, return the result of
 3070 instantiating the insertion context of the change.

3071
$$\begin{aligned} & (Mod\ p', Spn\ q') \rightarrow Mod \langle \$ \rangle mrgChgSpn\ p'\ q' \\ & (Spn\ p', Mod\ q') \rightarrow Mod \langle \$ \rangle mrgChgSpn\ q'\ p' \end{aligned}$$



(A) Call to `tryDel` succeeds; The *Bin* at the root can be deleted as it only overlaps with copies. `tryDel` returns the focus of the deletion and the part of the alignment *a* that overlaps with it.



(B) Call to `tryDel` fails; Although the *Bin* at the root could be deleted, the alignment *a* is changing the *42* present in the leaf. This is a conflict.

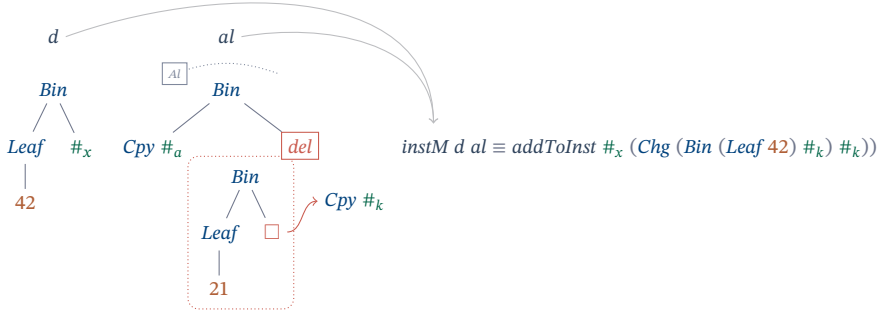
FIGURE 5.28: Two example calls to `tryDel`.

```

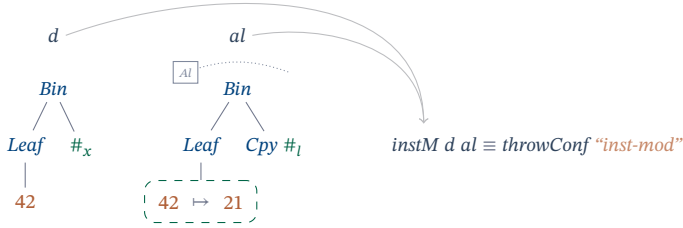
tryDel :: Zipper (CompoundCnstr κ fam x) (SFix κ fam) (Al κ fam (Chg κ fam)) x
  → Al κ fam (Chg κ fam) x
  → MergeM κ fam (Al κ fam (Chg κ fam) x, Al κ fam (Chg κ fam) x)
tryDel (Zipper z h) (Del (Zipper z' h'))
  | z ≡ z'   = return (h, h')
  | otherwise = throwConf "del-del"
tryDel (Zipper z h) (Spn rep) = case zipperRepZip z rep of
  Nothing → throwNotASpan
  Just r  → case partition (exElim isInR1) (repLeavesList r) of
    ([Exists (InL Refl :*: x)] , xs)
      | all isCpyL1 xs → return (h, x)
      | otherwise     → throwConf "del-spn"
    _                  → error "unreachable; zipRepZip invariant"
tryDel (Zipper _ _) _ = throwConf "del-mod"

```

FIGURE 5.29: Complete generic definition of the `tryDel` function.



(A) Call to *instM* succeeds and registers that the subtree identified by $\#_x$ has had its left child deleted, according to the alignment.



(B) Call to *instM* returns a conflict; The deletion context, *d*, wants to match against the value 42 but the alignment modifies it.

FIGURE 5.30: Two example calls to *instM*.

3072 The *mrgChgSpn* function, below, matches the deletion context of the *Chg* against
 3073 the spine and and returns a *P2Instantiate* instruction. The instantiation function *instM*,
 3074 exemplified in Figure 5.30 and defined in Figure 5.31, receives a deletion context and an
 3075 alignment and attempts to assign the variables in the deletion context to changes inside
 3076 the alignment. This is only possible, though, when the modifications in the spine occur
 3077 further from the root than the variables in the deletion context. Otherwise, we have a
 3078 conflict where some constructors flagged for deletion are also marked as modifications.

```

3079 mrgChgSpn :: (CompoundCnstr κ fam x) ⇒ Chg κ fam x → SRep (Al κ fam) (Rep x)
              → MergeM κ fam (Phase2 κ fam x)
mrgChgSpn p@(Chg dp _) spn = do
  instM dp (Spn spn)
  return (P2Instantiate p (Just (disalign (Spn spn))ins))

```

3080 The *Just* in the return value above indicate we must check that we will not introduce
 3081 extra duplications. In Figure 5.32 illustrates a case where failing to perform this check

```

instM :: (All Eq  $\kappa$ )  $\Rightarrow$  HolesMV  $\kappa$  fam at  $\rightarrow$  Al  $\kappa$  fam at  $\rightarrow$  MergeM  $\kappa$  fam ()
instM _ (Cpy _) = return ()
instM (Hole v) a = addToInst "inst-contr" v (disalign a)
instM _ (Mod _) = throwConf "inst-mod"
instM _ (Prm _ _) = throwConf "inst-perm"
-- Del ctx and spine must form a span; cannot reference different constructors or primitives.
instM x@(Prim _) d = when (x  $\neq$  (disalign d)del) throwNotASpan
instM (Roll r) (Spn s) = case zipSRep r s of
  Nothing  $\rightarrow$  throwNotASpan
  Just res  $\rightarrow$  void (repMapM ( $\lambda x \rightarrow$  uncurry' instM x  $\triangleright$  return x) res)
instM (Roll _) (Ins _) = throwConf "inst-ins"
instM (Roll _) (Del _) = throwConf "inst-del"

```

FIGURE 5.31: Implementation of `instM`, which receives a deletion context and an alignment and attempts to instantiate the variables in the deletion context with changes in the alignment.

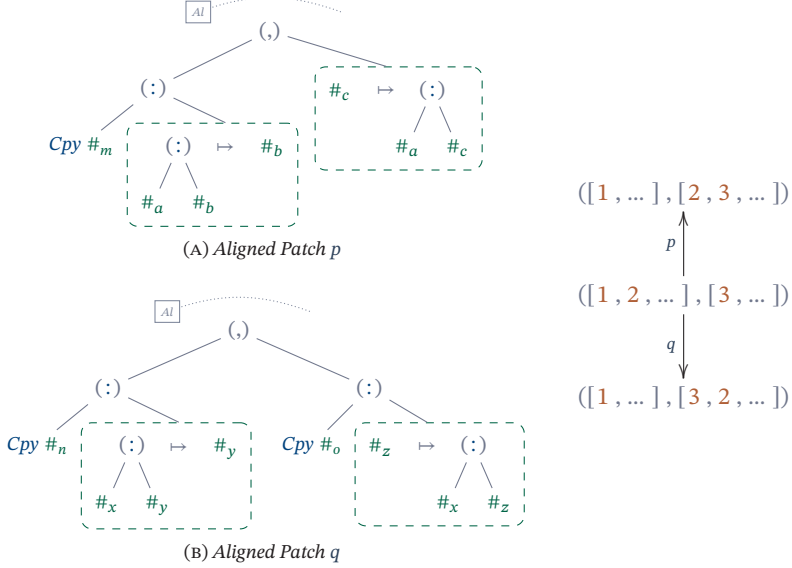
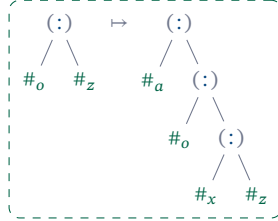


FIGURE 5.32: Example of two conflicting patches that move the same subtree into two different locations. The patches here are operating over pairs of lists.

would result in an erroneous duplication of the value 2. Matching the deletion context of $chg = Chg \#_c (\#_a : \#_c)$ against the spine $spn = Spn (Cpy \#_o : Chg \#_z (\#_x : \#_z))$ yields $\#_c$ equal to spn , which correctly identifies that the subtree at $\#_c$ was modified according to spn . The observation, however, is that the insertion context of chg mentions $\#_a$, which is a subtree that comes from outside the deletion context of chg . If we do not perform any further check and proceed naively, we would end up substituting $\#_c$ for $(disalign\ spn)_{del}$ and for $(disalign\ spn)_{ins}$ in chg_{del} and chg_{ins} , respectively, which would result in:



Since we know $\#_x \equiv \#_a$, which was registered when merging the left hand side of $(,)$, in Figures 5.32(A) and 5.32(B), it becomes clear that $\#_a$ was erroneously duplicated. Our implementation will reject this by checking that the set of subtrees that appear in the result of instantiating chg is disjoint from the set of subtrees moved by spn . \triangleleft *I dislike this aspect of this synchronization algorithm quite a lot, it feels unnecessarily complex and with no really good justification besides the example in Figure 5.32, which was distilled from real conflicts. I believe that further work would uncover a more disciplined way of disallowing duplications to be introduced.* \triangleright

Merging two spines is simple. We know they must reference the same constructor since the arguments to *merge* form a span. All that we have to do is recurse on the paired fields of the spines, point-wise:

```
(Spn p' , Spn q') → case zipSRep p' q' of
  Nothing → throwNotASpan
  Just r → Spn <$> repMapM (uncurry' mrgPhase1) r
```

Lastly, when the alignments in question are arbitrary modifications, we must try our best to reconcile these. We handle duplications differently than arbitrary modifications, they are easier to handle.

```
(Mod p' , Mod q') → Mod <$> mrgChgChg p' q'
```

A duplication or contraction is of the form $Chg \#_x \#_y$, where $\#_x$ or $\#_y$ occurs at least three times in the alignment at question. Three occurrences might seem arbitrary, but a metavariable must occur at least twice, and, when it occurs only twice the alignment algorithm would have marked it as a copy or a permutation. Merging duplications is

straightforward. When either one of p' or q' above are a duplication but the other is a change, we record how the tree was changed and move on.

```

3110   mrgChgDup :: Chg  $\kappa$  fam  $x \rightarrow$  Chg  $\kappa$  fam  $x \rightarrow$  MergeM  $\kappa$  fam (Phase2  $\kappa$  fam  $x$ )
3111   mrgChgDup dup@(Chg (Hole  $v$ )  $\_$ )  $q' =$  do
3112     addToInst "chg-dup"  $v$   $q'$ 
     return (P2Instantiate dup Nothing)

```

Finally, if p and q are not duplications, nor any of the cases previously discussed, then the best we can do is register equivalence of their domains – recall both patches being merged must form a span – and synchronize successfully when both changes are equal.

```

3117   mrgChgChg :: Chg  $\kappa$  fam  $x \rightarrow$  Chg  $\kappa$  fam  $x \rightarrow$  MergeM  $\kappa$  fam (Phase2  $\kappa$  fam  $x$ )
   mrgChgChg  $p' \ q' \mid$  isDup  $p' =$  mrgChgDup  $p' \ q'$ 
    $\mid$  isDup  $q' =$  mrgChgDup  $q' \ p'$ 
    $\mid$  otherwise = case unify  $p'_{\text{del}} \ q'_{\text{del}}$  of
     Left  $\_ \rightarrow$  throwNotASpan
     Right  $r \rightarrow$  onEqvs ( $M \cup r$ )  $\triangleright$  return (P2TestEq  $p' \ q'$ )

```

Once the first pass is done and we have collected information about how each subtree has been changed and potential subtree equivalences we might have discovered. The next step is to synthesize this information into two maps: a deletion map that informs us what a subtree *was* and an insertion map that informs us what a subtree *became*, so we can perform the *P2Instantiate* and *P2TestEq* instructions.

SECOND PHASE. The second phase starts with splitting *inst* and *eqvs*, which requires some attention. For example, imagine there exists an entry in *inst* that assigns $\#_x$ to *Chg (Hole $\#_y$) (:42 (Hole $\#_y$))*, this tells us that the tree identified by $\#_x$ is the same as the tree identified by $\#_y$, and it became (:42 $\#_y$). Now suppose that $\#_x$ was duplicated somewhere else, and we come across an equivalence that says $\#_y \equiv \#_x$. We cannot simply insert this equivalence into *inst* because the merge algorithm made the decision to remove all occurrences of $\#_x$, not of $\#_y$, even though they identify the same subtree. This is important to ensure we produce patches that can be applied. \triangleleft *This is yet another aspect I am unsatisfied with and would like to see a more disciplined approach. Will have to be future work, nevertheless.* \triangleright

The *splitDelInsMaps* function is responsible for synthesizing the information gathered in the first pass of the synchronization algorithm. First we split *inst* into the deletion and insertion components of each of its points. Next, we partition the equivalences into rigid equivalences, of the form $(\#_v, t)$ where t has no holes, and non-rigid equivalences. The rigid equivalences are added to both deletion and insertion maps, but the non-rigid ones, $(\#_v, t)$, are only added when there is no information about the $\#_v$ in the map

3139 and, if $t \equiv \#_u$, we also check that there is no information about $\#_u$ in the map. Lastly,
 3140 after these have been added to the map, we call *minimize* to produce an idempotent sub-
 3141 stitution we can use for phase two. If an occurs-check error is raised, this is forwarded
 3142 as a conflict.

```

3143 type Subst2  $\kappa$  fam = (Subst  $\kappa$  fam (Metavar  $\kappa$  fam) , Subst  $\kappa$  fam (Metavar  $\kappa$  fam))
splitDelInsMaps :: MergeState  $\kappa$  fam  $\rightarrow$  Either [Exists (Metavar  $\kappa$  fam)] (Subst2  $\kappa$  fam)
splitDelInsMaps (MergeState iot eqvs) = do
  let e' = splitEqvs eqvs
  d  $\leftarrow$  addEqvsAndSimpl (map (exMap  $\cdot_{\text{del}}$ ) inst) e'
  i  $\leftarrow$  addEqvsAndSimpl (map (exMap  $\cdot_{\text{ins}}$ ) inst) e'
  return (d , i)

```

3144 After computing the insertion and deletion maps, which inform us how each identi-
 3145 fied subtree was modified, we start a second pass over the result of the first pass and
 3146 execute the necessary instructions.

```

phase2 :: Subst2  $\kappa$  fam  $\rightarrow$  Phase2  $\kappa$  fam at  $\rightarrow$  MergeM  $\kappa$  fam (Chg  $\kappa$  fam at)
phase2 di (P2TestEq ca cb) = isEqChg di ca cb
phase2 di (P2Instantiate chg Nothing) = return (refineChg di chg)
phase2 di (P2Instantiate chg (Just i)) = do
  es  $\leftarrow$  gets eqs
  case getCommonVars (substApply es chgins) (substApply es i) of
    []  $\rightarrow$  return (refineChg di chg)
    xs  $\rightarrow$  throwConf ("mov-mov "  $\#$  show xs)

```

3148 The *getCommonVars* computes the intersection of the variables in two *Holes*, which
 3149 is used to forbid subtrees to be moved in two different ways.

3150 Refining changes according to the inferred information is straightforward, all we
 3151 must do is apply the deletion map to the deletion context and the insertion map to the
 3152 insertion context.

```

3153 refineChg :: Subst2  $\kappa$  fam  $\rightarrow$  Chg  $\kappa$  fam at  $\rightarrow$  Chg  $\kappa$  fam at
refineChg (d , i) (Chg del ins) = Chg (substApply d del) (substApply i ins)

```

3154 When deciding whether two changes are equal, its also important to refine them
 3155 first, since they might be α -equivalent.

```

isEqChg :: Subst2  $\kappa$  fam  $\rightarrow$  Chg  $\kappa$  fam at  $\rightarrow$  Chg  $\kappa$  fam at  $\rightarrow$  Maybe (Chg  $\kappa$  fam at)
isEqChg di ca cb = let ca' = refineChg di ca
  cb' = refineChg di cb
  in if ca'  $\equiv$  cb' then Just ca' else Nothing

```

3157 The merging algorithm presented in this section is involved. It must deal with a
3158 number of corner cases and use advanced techniques to do so generically. Most of the
3159 difficulties come from having to deal with arbitrary duplications and contractions. If we
3160 instead chose to use only linear patches, that is, patches where each metavariable must
3161 be declared and used exactly once, the merge algorithm could be simplified.

3162 5.4 DISCUSSION AND FURTHER WORK

3163 With `hdiff` we have seen that a complete detachment from edit-scripts enables us to
3164 define a computationally efficient differencing algorithm and how the notion of *change*
3165 coupled with a simple notion of composition gives a sensible algebraic structure. The
3166 patch datatype in `hdiff` is more expressive than edit-script based approaches, it en-
3167 ables us to write transformations involving arbitrary permutations and duplications. As
3168 a consequence, we have a more involved merge algorithm. For one, we cannot easily
3169 generalize our three-way merge to n -way merge. More importantly, though, there are
3170 subtleties in the algorithm that arose purely from practical necessities. Our posterior
3171 empirical evaluation (Chapter 6) does indicate that the best success ratio comes from
3172 merging linear patches – where metavariables occur exactly twice, obtained with the
3173 *Patience* extraction mode. This does suggest that the soft-spot in the design space might
3174 well be allowing arbitrary permutations, enabling a fast differencing algorithm, but for-
3175 bidding arbitrary duplications and contractions, which could enable a simpler merging
3176 algorithm. Besides the merging algorithm, we will discuss a number of other important
3177 aspects that were left as future work and would need to be addressed to bring `hdiff`
3178 from a prototype to a production tool.

3179 REFINING MATCHING AND SHARING CONTROL

3180 The matching engine underlying `hdiff` uses hashes indiscriminately, all information
3181 under a subtree is used to compute its hash, which can be undesirable. Imagine a parser
3182 that annotates its resulting AST with source-location tokens. This means that we would
3183 not be able to recognize permutations of statements, for example, since both occurrences
3184 would have different source-location tokens and, consequently, different hashes.

3185 This issue goes hand in hand with deciding which parts of the tree can be shared and
3186 up until which point. For example, we probably never want to share local statements
3187 outside their scope. Recall we avoided this issue by restricting whether a subtree could
3188 be shared or not based on its height. This was a pragmatic design choice that enabled us
3189 to make progress but it is a work-around at its best.

3190 Salting the hash function of *preprocess* is not an option for working around the issue
3191 of sharing control. If the information driving the salt function changes, none of the sub-
3192 trees under there can be shared again. To illustrate this, suppose we push scope names

into a stack with a function *intrScope* $:: SFix \kappa fam at \rightarrow Maybe String$, which would be supplied by the user. It returns a *Just* whenever the datatype in question introduces a scope. The *const Nothing* function works as a default value, meaning that the mutually recursive family in question has no scope-dependent naming. A more interesting *intrScope*, for some imaginary mutually recursive family, is given below.

```

intrScope m@(Module ...) = Just (moduleName m)
intrScope f@(FunctionDecl ...) = Just (functionName f)
intrScope _ = Nothing

```

With *intrScope* as above, we could instruct the *preprocess* to push module names and function names every time it traverses through one such element of the family. For example, preprocessing the pseudo-code below would mean that the hash for a inside *fib* would be computed with $[“m”, “fib”]$ as a salt; but a inside *fat* would be computed with $[“m”, “fat”]$ as a salt, yielding a different hash.

```

module m
  fib n = let a = 0; b = 1; ...
  fat n = let a = 0; ...

```

This will work out well for many cases, but as soon as a change altered any information that was being used as a salt, nothing could be shared anymore. For example, if we rename *module m* to *module x*, the source and destination would contain no common hashes, since we would have used $[“m”]$ to salt the hashes for the source tree, but $[“x”]$ for the destination, yielding different hashes.

This problem is twofold, however. Besides identifying the algorithmic means to ensure *hdiff* could be scope-aware and respect said scopes, we must also engineer an interface to enable the user to easily define said scopes. I envisioned a design with a custom version of the *generics-simplistic* library, with an added alias for the identity functor that could receive special treatment, for example:

```

newtype Scoped f = Scoped { unScoped :: f }
data Decl = ImportDecl ...
          | FunDecl String [ParmDecl] (Scoped Body)
          ...

```

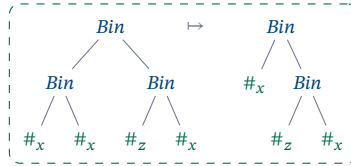
This would mean that when inspecting and pattern matching on *SRep* throughout our algorithms, we could treat *scoped* types differently.

We reiterate that if there is a solution to this problem, it certainly will not use a modification of the matching mechanism: if we use scope names, renamings will cause problems; if we use the order which scopes have been seen (De Bruijn-like), permutations will cause problems. Controlling on the height of the trees and minimizing this

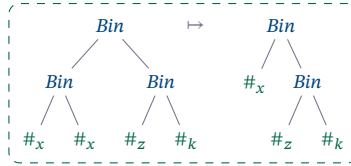
issue was the best option to move forward in an early stage. Unfortunately, I did not have time to explore how scope graphs [75] could help us here, but it is certainly a good place to start looking. It might be possible to use scope graphs to write a more intricate *close* function, that will properly break sharing where necessary, for example.

EXTRACTION METHODS, *BEST* PATCH AND EDIT-SCRIPTS

We have presented three extraction methods, which we called *NoNested*, *ProperShare* and *Patience*. Computing the diff between two trees using different extraction methods can produce different patches. Certainly there can be more extraction methods. One such example that I never had the time to implement was a refinement of *ProperShare*, aimed at breaking the sharing introduced by it. The idea was to list the the metavariables that appear in the deletion and insertion context and compute the LCS between these lists. The location of copies enable us to break sharing and introduce new metavariables. For example, take the change below.



The list of metavariables in the deletion context is $[\#_x, \#_x, \#_z, \#_x]$, but in the insertion context we have $[\#_x, \#_z, \#_x]$. Computing the longest common subsequence between these lists yields $[Del\ x, Cpy, Cpy, Cpy]$. The first *Del* suggests a contraction is really necessary, but the last copy shows that we could *break* the sharing by renaming $\#_x$ to $\#_k$, for example. This would essentially transform the change above into:



The point is that the copying of $\#_z$ can act as a synchronization point to introduce more variables, forget some sharing constraints, and ultimately enlarge the domain of our patches.

Forgetting about sharing is just one example of a different context extraction mechanism and, without a formal notion about when a patch is *better* than another, its impossible to make a decision about which context extraction should be used. Our experimental results suggest that *Patience* yields patches that merge successfully more often, but this is far from providing a metric on patches, like the usual notion of cost does for edit-scripts.

3249 RELATION TO EDIT-SCRIPTS. Another interesting aspect that I would have liked to look
3250 at is the relation between our *Patch* datatype and traditional edit-scripts. The idea of
3251 breaking sharing above can be used to translate our patches to an edit-script. Some early
3252 experiments did show that we could use this method to compute approximations of the
3253 least-cost edit-script in linear time. Given that the minimum cost edit-script takes nearly
3254 quadratic time [10], it might be worth looking into how good an approximation we might
3255 be able to compute in linear time.

3256 FORMALIZATIONS AND GENERALIZATIONS

3257 Formalizing and proving properties about our *diff* and *merge* functions was also one of
3258 my priorities. As it turns out, the extensional nature of *Patch* makes for a difficult Agda
3259 formalization, which is the reason this was left as further work.

3260 The value of a formalization goes beyond enabling us to prove important proper-
3261 ties. It also provides a laboratory for generalizing aspects of the algorithms. Two of
3262 those immediately jump to mind: generalizing the merge function to merge n patches
3263 and generalizing alignments insertions and deletions zippers to be of arbitrary depth,
3264 instead of a single layer. Finally, a formalization also provides important value in better
3265 understanding the merge algorithm.

3266



3267

3268 EXPERIMENTS

3269 Throughout this thesis we have presented two approaches to structural differencing. In
3270 Chapter 4 we saw `stdiff`, which although unpractical, provided us with important in-
3271 sights into the representation of patches. These insights and experience led us to develop
3272 `hdiff`, Chapter 5, which improved upon the previous approach with a more efficient
3273 *diff* function at the expense of the simplicity of the merge algorithm: the *merge* function
3274 from `hdiff` is much more involved than that of `stdiff`.

3275 In this chapter we evaluate our algorithms on real-world conflicts extracted from
3276 GitHub and analyze the results. We are interested in performance measurements and
3277 synchronization success rates, which are central factors to the applicability of structural
3278 differencing in the context of software version control.

3279 To conduct the aforementioned evaluation we have extracted a total of 12 552 usable
3280 datapoints from GitHub. They have been obtained from large public repositories storing
3281 code written in Java, JavaScript, Python, Lua and Clojure. The choice of programming
3282 languages was motivated by the availability of parsers, with the exception of Clojure,
3283 where we borrowed a parser from a MSc thesis [33]. More detailed information about
3284 data collection is given in Section 6.1.

3285 The evaluation of `stdiff` has fewer datapoints than `hdiff` for the sole reason that
3286 `stdiff` requires the `generics-mrsop` library, which triggers a memory leak in GHC¹
3287 when used with larger abstract syntax trees. Consequently, we could only evaluated
3288 `stdiff` over the Clojure and Lua subset of our dataset.

¹<https://gitlab.haskell.org/ghc/ghc/issues/17223> and <https://gitlab.haskell.org/ghc/ghc/issues/14987>

6.1 DATA COLLECTION

Collecting files from GitHub can be done with the help of some bash scripting. The overall idea is to extract the merge conflicts from a given repository by listing all commits c with more than two parents, recreating the repository at the state immediately previous to c then attempting to call `git merge` at that state.

Our script improves upon the script written by a master student [33] by making sure to collect the file that a human committed as the resolution of the conflict, denoted `M.lang`. To collect conflicts from a repository, then, all we have to do is run the following commands at its root.

- List each commit c with at least two parents with `git rev-list --merges`.
- For each commit c as above, let its parents be p_0 and ps ; checkout the repository at p_0 and attempt to `git merge --no-commit ps`. The `--no-commit` switch is important since it gives us a chance to inspect the result of the merge.
- Next we parse the output of `git ls-files --unmerged`, which provides us with the three *object-ids* for each file that could not be automatically merged: one identifier for the common ancestor and one identifier for each of the two diverging replicas.
- Then we use `git cat-file` to get the files corresponding to each of the *object-ids* gathered on the previous step. This yields three files, `O.lang`, `A.lang` and `B.lang`. Lastly, we use `git show` to save the file `M.lang` that was committed by a human resolving the conflict.

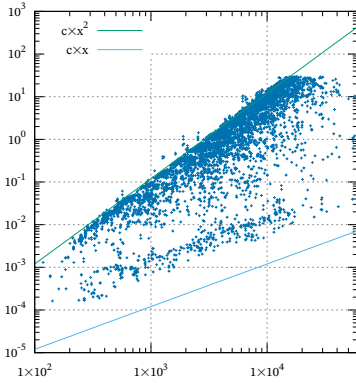
After running the steps above for a number of repositories, we end up with a list of folders containing a merge conflict that was solved manually. Each of these folders contain a span $A \leftarrow O \rightarrow B$ and a file M which is the human-produced result of synchronizing A and B . We refer the reader to the full code for more details (Appendix A). Overall, we acquired 12 552 usable conflicts – that is, we were able to parse the four files with the parsers available to us – and 2 771 conflicts where at least one file yielded a parse error. Table 6.1 provides the distribution of datapoints per programming language and displays the number of conflicts that yielded a parse error. These parse errors are an inevitable consequence of using off-the-shelf parsers on an existing dataset. The parseable conflicts have been compiled into a publicly available dataset [67].

6.2 PERFORMANCE

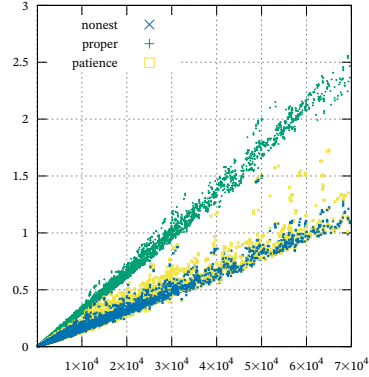
To measure the performance of the *diff* functions in both approaches we computed four patches per datapoint, namely: `diff O A`, `diff O B`, `diff O M` and `diff A B`.

Language	Repositories	Parseable Conflicts	Non-parseable Conflicts
Clojure	31	1 213	16
Java	19	2 901	851
JavaScript	28	3 392	965
Lua	27	748	91
Python	27	4 298	848
Totals	132	12 552	2 771

TABLE 6.1: Distribution of datapoints within our dataset [67]. The repositories were chosen manually by searching each respective language in GitHub. Our criteria for selecting repositories to mine was based on number of forks and commits, in an attempt to maximize pull requests.



(A) Runtimes from `stdiff` shown in a log-log plot. The lines illustrate the behavior of `stdiff` being between linear and quadratic



(B) Runtimes from `hdiff` shown in a linear plot.

FIGURE 6.1: Performance measurements of `stdiff` and `hdiff` differencing functions. The vertical axis represents seconds and the horizontal axis has the sum of the number of constructors in the source and destination trees.

3323 Whilst computing patches we limited the memory usage to 8GB and runtime to 30s.
 3324 If a call to *diff* used more than the available temporal and spacial resources it was auto-
 3325 matically killed. We ran both *stdiff* and *hdiff* on the same machine, yet, we stress
 3326 that the absolute values are of little interest. The real take away from this experiment
 3327 is the empirical validation of the complexity class of each algorithm. The results are
 3328 shown in Figure 6.1 and plot the measured runtime against the sum of the number of
 3329 constructors in the source and destination trees.

3330 Figure 6.1(A) illustrated the measured performance of the differencing algorithm in
 3331 *stdiff*, our first structural differencing tool, discussed in Section 4.3.2. Let *fa* and *fb* be
 3332 the files being differenced, we have only timed the call to *diff fa fb* – which excludes pars-
 3333 ing. Note that most of the time, *stdiff* exhibits a runtime proportional to the square of
 3334 the input size. That was expected since it relies on a quadratic algorithm to annotate the
 3335 trees and then translate the annotated trees into *Patch_{ST}* over a single pass. Out of the
 3336 8428 datapoints where we attempted to time *stdiff* in order to produce Figure 6.1(A),
 3337 913 took longer than thirty seconds and 929 used more than 8GB of memory. The rest
 3338 have been plotted in Figure 6.1(A).

3339 Figure 6.1(B) illustrates the measured performance of the differencing algorithm un-
 3340 derlying *hdiff*, discussed in Section 5.1.4. We have plotted each of the context extrac-
 3341 tion techniques described in 5.1.4.2. The linear behavior is evident and in general, an
 3342 order of magnitude better than *stdiff*. We do see, however, that the proper context
 3343 extraction is slightly slower than *nonest* or *patience*. Finally, only 14 calls timed-out
 3344 and none used more than 8GB of memory.

3345 Measuring performance of pure Haskell code is subtle due to its lazy evaluation se-
 3346 mantics. We have used the *time* auxiliary function below. We based ourselves on the
 3347 *timeit* package, but adapted it to fully force the evaluation of the result of the action,
 3348 with the *deepseq* method and force its execution with the bang pattern in *res*, ensuring
 3349 the thunk is fully evaluated.

```

time :: (NFData a) => IO a -> IO (Double, a)
time act = do t1 <- getCPUTime
              result <- act
              let ! res = result `deepseq` result
              t2 <- getCPUTime
              return (fromIntegral (t2 - t1) * 1e-12, res)
3350

```

3351 6.3 SYNCHRONIZATION

3352 While the performance measurements provide some empirical evidence that *hdiff* is
 3353 in the right complexity class, the synchronization experiment, discussed in this section,

Language	<i>success</i>	(ratio)	<i>mdif</i>	(ratio)	total ratio	<i>conf</i>	<i>t/o</i>
Clojure	184	(0.15)	211	(0.17)	0.32	818	0
Java	978	(0.34)	479	(0.16)	0.5	1 443	1
JavaScript	1 046	(0.30)	274	(0.08)	0.38	2 062	10
Lua	185	(0.25)	101	(0.14)	0.39	462	0
Python	907	(0.21)	561	(0.13)	0.34	2 829	1
<i>Total</i>	3 300	(0.26)	1 626	(0.13)	0.39	7 614	12

TABLE 6.2: *Best synchronization success rate per language. No apply-fail was encountered in the entire dataset and the number of timeouts was negligible.*

aims at establishing a lower bound on the number of conflicts that could be solved in practice.

The synchronization experiment consists of attempting to merge the $A \leftarrow O \rightarrow B$ span for every datapoint. If `hdiff` produces a patch with no conflicts, we apply it to O and compare the result against M , which was produced by a human. There are four possible outcomes, three of which we expect to see and one that would indicate a more substantial problem. The three outcomes we expect to see are: *success*, which indicates the merge was successful and was equal to that produced by a human; *mdif* which indicates that the merge was successful but produced a different than the manual merge; and finally *conf* which means that the merge was unsuccessful. The other possible outcome comes from producing a patch that *cannot* be applied to O , which is referred to as *apply-fail*. Naturally, timeout or out-of-memory exceptions can still occur and fall under *other*. The merge experiment was capped at 45 seconds of runtime and 8GB of virtual memory.

The distinction between *success* and *mdif* is important. Being able to merge a conflict but obtaining a different result from what was committed by a human does not necessarily imply that either result is wrong. Developers can perform *more or fewer* modifications when committing M . For example, Figure 6.2 illustrates an example distilled from our dataset which the human performed an extra operation when merging, namely adapting the *sheet* field of one replica. It can also be the case that the developer made a mistake which was fixed in a later commit. Therefore, a result of *mdif* in a datapoint does not immediately indicate the wrong behavior of our merging algorithm. The success rate, however, provides us with a reasonable lower bound on the number of conflicts that can be solved automatically, in practice.

Given the multitude of dials we can adjust in `hdiff`, we have run the experiment with each combination of extraction method (*Patience*, *NoNested*, *ProperShare*), local or

Language	Mode	Height	success	(ratio)	mdif	(ratio)	conf	t/o
Clojure	<i>Patience</i>	1	184	(0.15)	211	(0.17)	818	0
	<i>NoNested</i>	3	149	(0.12)	190	(0.16)	874	0
	<i>ProperShare</i>	9	92	(0.08)	84	(0.07)	1 037	0
Java	<i>Patience</i>	1	978	(0.34)	479	(0.16)	1 443	1
	<i>NoNested</i>	3	924	(0.32)	509	(0.18)	1 467	1
	<i>ProperShare</i>	9	548	(0.19)	197	(0.07)	2 155	1
JavaScript	<i>Patience</i>	1	1 046	(0.30)	274	(0.08)	2 062	10
	<i>NoNested</i>	3	991	(0.29)	273	(0.08)	2 124	4
	<i>ProperShare</i>	9	748	(0.22)	116	(0.03)	2 508	20
Lua	<i>Patience</i>	3	185	(0.25)	101	(0.14)	462	0
	<i>NoNested</i>	3	171	(0.23)	110	(0.15)	467	0
	<i>ProperShare</i>	9	86	(0.11)	29	(0.04)	633	0
Python	<i>Patience</i>	1	907	(0.21)	561	(0.13)	2 829	1
	<i>NoNested</i>	3	830	(0.19)	602	(0.14)	2 865	1
	<i>ProperShare</i>	9	446	(0.10)	223	(0.05)	3 627	2

TABLE 6.3: Best results for each extraction mode. The height column indicates the minimum height a subtree must have to qualify for sharing, configured with the `--min-height` option. All of the above results were obtained with locally-scoped patches, globally-scoped success rates were consistently lower than their locally-scoped counterpart.

<pre>d={name='A', sheet='a' ,name='B', sheet='b' ,name='C', sheet='c'}</pre> <p style="text-align: center;">(A) Replica A</p>	<pre>d={name='A', sheet='path/a' ,name='B', sheet='path/b' ,name='X', sheet='path/x' ,name='C', sheet='path/c'}</pre> <p style="text-align: center;">(B) Replica B</p>
<pre>d={name='A', sheet='path/a' ,name='B', sheet='path/b' ,name='C', sheet='path/c'}</pre> <p style="text-align: center;">(C) Common ancestor, \emptyset</p>	
<pre>d={name='A', sheet='a' ,name='B', sheet='b' ,name='X', sheet='x' ,name='C', sheet='c'}</pre> <p style="text-align: center;">(D) Merge produced by a human</p>	<pre>d={name='A', sheet='a' ,name='B', sheet='b' ,name='X', sheet='path/x' ,name='C', sheet='c'}</pre> <p style="text-align: center;">(E) Merge produced by <i>hdiff</i></p>

FIGURE 6.2: Example distilled from *hawkthorne-server-lua*, commit 60eba8. One replica introduced entries in a dictionary where another transformed a system path. The *hdiff* tool did produce a correct merge given, but this got classified as *mdif*.

global metavariable scoping and minimum sharing height of 1, 3 and 9. Table 6.3 shows the combination of parameters that yielded the most successes per extraction method, the column for scoping is omitted because local scope outperformed global scoping in all instances. Table 6.2 shows only the highest success rate per language.

The varying true success rates seen in Table 6.3 are to be expected. Different parameters used with *hdiff* yield different patches, which might be easier or harder to merge. Out of the datapoints that resulted in *mdif* we have manually analyzed 16 randomly selected cases. We witnessed that 13 of those *hdiff* behaved as we expect, and the *mdif* result was attributed to the human performing more operations than a structural merge would have performed, as exemplified in Figure 6.2, which was distilled from the manually analyzed cases. We will shortly discuss two cases, illustrate in Figures 6.3 and 6.4, where *hdiff* behaved unexpectedly.

It is worth noting that even though 100% success rate is unachievable – some conflicts really come from a subtree being modified in two distinct ways and inevitably require human intervention – the results we have seen are very encouraging. In Table 6.2 we see that *hdiff* produces a merge in at least 39% of datapoints and the majority of the time, it matches the handmade merge.

The cases where *the same* datapoint yields a true success and a *mdif*, depending on which extraction method was used, are interesting. Let us look at two complimentary examples (Figures 6.3 and 6.4) that were distilled from these contradicting cases.

```

class Class {
  public int f(int x)
  { F(x*y); }
  public int g(int x)
  { G(x+2); }
}
(A) A.java

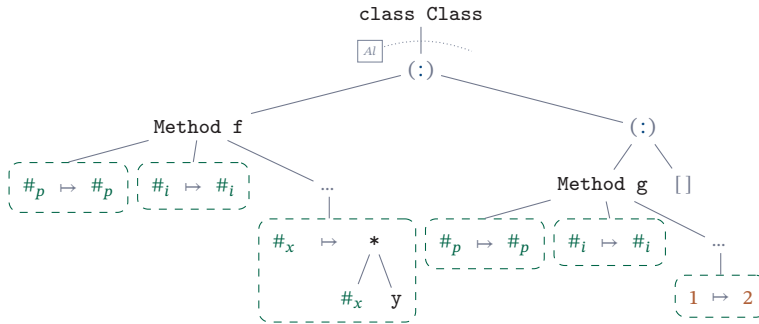
class Class {
  public int f(int x)
  { F(x); }
  public int g(int x)
  { G(x+1); }
}
(B) O.java

class Class {
  public int f(int x)
  { F(x); }
  public static int g(int x)
  { G(x+1); }
}
(C) B.java

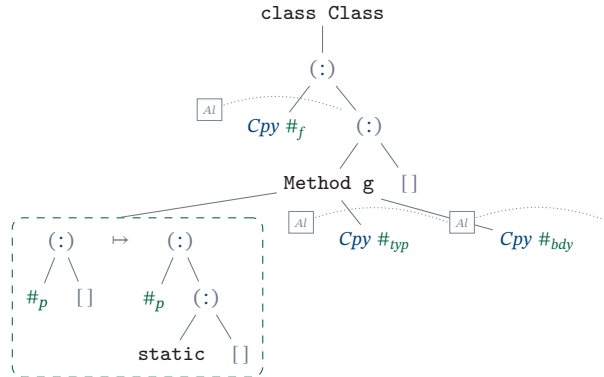
class Class {
  public int f(int x)
  { F(x*y); }
  public static int g(int x)
  { G(x+2); }
}
(D) Expected merge, computed with Patience

class Class {
  public static int f(int x)
  { F(x*y); }
  public static int g(int x)
  { G(x+2); }
}
(E) Incorrect merge, computed with NoNest

```



(F) Simplified illustration of patch computed with `hdiff -d nonest {O,A}.java`; The sharing of $\#_p$ reflects the sharing of the list of method modifiers.



(G) Simplified illustration of patch computed with `hdiff -d nonest {O,B}.java`, note how each copy happens inside its own scope

FIGURE 6.3: Example distilled from `cas`, commit 035eae3, where *Patience* merges with a true success but *NoNest* merges with *mdif*, and, in fact, replicates the `static` modifier incorrectly.

<pre> class Class { String S = C.g(); void m () { return; } void o (int l); void p (); } </pre>	<pre> class Class { void m () { C.q.g(); return; } void n (); void o (); void p (); } </pre>	<pre> class Class { void m () { C.q.g(); return; } void n (); void o (); void X (); void p (); } </pre>
(A) <i>A.java</i>	(B) <i>O.java</i>	(C) <i>B.java</i>

<pre> class Class { String S = C.g(); void m () { return; } void o (int l); void X (); void p (); } </pre>	<pre> class Class { String S = C.g(); void X (); void m () { return; } void o (int l); void p (); } </pre>
(D) <i>Expected merge, computed with</i> <i>NoNested</i>	(E) <i>Incorrect merge, computed</i> <i>with Patience</i>

FIGURE 6.4: Example distilled from *spring-boot*, commit 0074e9, where *NoNested* merges with a true success but *Patience* merges with *mdiff* since it inserts the declaration of *X* in the wrong place.

Figure 6.3 shows an example where merging patches extracted with *Patience* returns the correct result, but merging patches extracted with *NoNest* does not. Because replica A modified the definition of *f*, the entire declaration of *f* cannot be copied, and it is placed inside the same scope (alignment) as the definition of *g* since they share a name, *x*. They also share, however, the list of method modifiers, which in this case is *public*. When B modifies the list of modifiers of method *g* by appending *static*, the merge algorithm replicates this change to the list of modifiers of *f*, since the patch wrongly believes both lists represent *the same list*. Merging with *Patience* does not witness the problem since it will not share *x* nor the modifier list, since these occur more than once in the deletion and insertion context of both *hdiff O A* and *hdiff O B*.

Figure 6.4, on the other hand, shows an example where merging patches extracted with *NoNested* succeeds, but *Patience* inserts a declaration in an unexpected location. Upon further inspection, however, the reason for the diverging behavior becomes clear. When differencing A and O under *Patience* context extraction, the empty bodies (which are represented in the Java AST by *MethodBody Nothing*) of the declarations of *n* and *o* are not shared. Hence, the alignment mechanism wrongly identifies that *both* *n* and *o*. Moreover, because *C.g()* is uniquely shared between the definition of *m* and *S*, the patch identifies that *void m...* became *String S...* Finally, the merge algorithm then transforms *void m* into *String S*, but then sees two deletions, which trigger the deletion of *n* and *o* from the spine. The next instruction is the insertion of *X*, resulting in the non-intuitive placement of *X* in the merge produced with *Patience*. When using

	<i>not-eq</i>	<i>inst-mod</i>	<i>del-spn</i>	<i>ins-ins</i>	<i>inst-ins</i>	<i>inst-del</i>	Others
Amount	7904	5052	2144	1892	868	357	506
Ratio	0.42	0.27	0.11	0.1	0.05	0.02	0.03

TABLE 6.4: *Distribution of conflicts observed by running `hdiff` over our dataset [67]. The first row displays the number of times that `throwConf` was called with which label.*

3421 *NoNested*, however, the empty bodies get all shared through the code and prevent the
 3422 detection of a deletion by the alignment algorithm. It is worth noting that just because
 3423 Java does not order its declarations, this is not acceptable behavior since it could produce
 3424 invalid source files in a language like Agda, where the order of declarations matter, for
 3425 example.

3426 The examples in Figures 6.3 and 6.4 illustrate an inherent difficulty of using naive
 3427 structured differencing over structures with complex semantics, such as source-code.
 3428 On the one hand sharing method modifiers triggers undesired replication of a change.
 3429 On the other, the lack of sharing of empty method bodies makes it difficult to place an
 3430 insertion in its correct position.

3431 When `hdiff` returned a patch with conflicts, that is, we could *not* successfully solve
 3432 the merge, we recorded the class of conflicts we observed. Table 6.4 shows the distri-
 3433 bution of each conflict type throughout the dataset. Note that a patch resulting from a
 3434 merge can have multiple conflicts. This information is useful for deciding which aspects
 3435 of the merge algorithm can yield better results.

3436 6.3.1 THREATS TO VALIDITY

3437 The synchronization experiment is encouraging, but before drawing conclusions how-
 3438 ever, we must analyze our assumptions and setting and preemptively understand which
 3439 factors could also be influencing the numbers.

3440 We are differencing and comparing objects *after* parsing. This means that comments
 3441 and formatting data was completely ignored. In fact, preliminary evaluations showed
 3442 that a vastly inferior success rate results from incorporating and considering source-
 3443 location tokens in the abstract syntax tree. This is expected since the insertion of a single
 3444 empty line, for example, will change the hashes that identify all subsequent elements of
 3445 the abstract syntax and stop them from being shared. The source-location tokens essen-
 3446 tially make the transformations that happen further down the file to be undetected using
 3447 `hdiff`. Although `stdiff` would not suffer from this problem, it is already impractical
 3448 by itself.

Our decision of disconsidering formatting, comments and source-location tokens is twofold. First, the majority of the available parsers does not include said information. Secondly, if we had considered all that information in our merging process, the final numbers would not inform us about how many code transformations are *disjoint* and could be automatically merged.

Another case worth noting is that although we have not found many cases where `hdiff` performed a wrong merge, Figures 6.3 and 6.4 showcases two such cases, hence, it is important to take the aggregate success rate with a grain of salt. There exists a probability that some of the *mdif* cases are false positives, that is, `hdiff` produced a merge but it performed the wrong operation.

Finally, one can also argue we have not considered conflicts that arise from rebasing, as these are not observed in the git history. This does not necessarily make a threat to validity, but indeed would have given us more data. That being said, we would only be able to recreate rebases done through GitHub web interface. The rebases done on the command line are impossible to recreate.

6.4 DISCUSSION

This chapter provided an empirical evaluation of our methods and techniques. We observed how `stdiff` is at least one order of magnitude slower than `hdiff`, confirming our suspicion of it unusable in practice. Preliminary synchronization experiments done with `stdiff` over the same data revealed a comparatively small success rate. Around 15% of the conflicts could be solved, out of which 60% did match what a human did.

The measurements for `hdiff`, on the other hand, gave impressive results. Even with all the overhead introduced by generic programming and an unoptimized algorithm, we can still compute patches almost instantaneously. Moreover, it confirms our intuition that the differencing algorithm underlying `hdiff` is in fact linear.

The synchronization results for `hdiff` are encouraging. A proper calculation of the probability that a conflict encountered in GitHub could be solved automatically is involved and out of the scope of this thesis. Nevertheless, we have observed that 39% of the conflicts in our dataset could be solved by `hdiff` and 66% of these solutions did match what a human performed.

An interesting observation that comes from the synchronization experiment, Table 6.3, is that the best merging success rate for all languages used the *Patience* context extraction – only copying subtrees that occur uniquely. This suggests that it might be worthwhile to forbid duplication and contractions on the representation level and work on a merging algorithm that enjoys the precondition that each metavariable occurs only twice. This simplification could enable us to write a simpler merging algorithm and an Agda model, which can then be used to prove important properties about our algorithms.

3486



3487

3488 DISCUSSION

3489 Even though the main topic of this thesis is *structural differencing*, a significant part
3490 of the contribution lies in field of generic programming. The two libraries we wrote
3491 make it possible to use powerful generic programming techniques over larger classes of
3492 datatypes than what was previously available. In particular, defining the generic inter-
3493 pretation as a cofree comonad and a free monad combined in a single datatype is very
3494 powerful. Being able to annotate and augment datatypes, for example, was paramount
3495 for scaling our algorithms.

3496 On *structural differencing*, we have explored two preliminary approaches. A first
3497 method, `stdiff`, was presented in Chapter 4 and revealed itself to be unpractical due
3498 to poor performance. The second method, `hdiff`, introduced in Chapter 5, has shown
3499 much greater potential. Empirical results were discussed in Chapter 6.

3500 7.1 THE FUTURE OF STRUCTURAL DIFFERENCING

3501 The larger picture of structural differencing is more subtle, though. It is not because
3502 our preliminary prototype has shown good results that we are ready to scale it to be the
3503 next `git merge`. There are three main difficulties in applying structural differencing
3504 to source-code with the objective of writing better merge algorithms:

- 3505 a) How to properly handle formatting and comments of source code: should the AST
3506 keep this information? If so, the tree matching must be adapted to cope with this.
3507 Two equal trees must be matched regardless of whether or not they appeared with
3508 a different formatting in their respective source files.

- 3509 b) How to ensure that subtrees are only being shared within their respective scope
3510 and, equally importantly, how to specify which datatypes of the AST are affected
3511 by scopes.
- 3512 c) When merging fails, returning a patch with conflicts, a human must interact with
3513 the tool and solve the conflicts. What kind of interface would be suitable for that?
3514 Further ahead, comes the question of automatic conflict solving domain-specific
3515 languages. Could we configure the merge algorithm to always chose higher version
3516 numbers, for example, whenever it finds a conflict in, say, a config file?

3517 Fixing the obstacles above in a generic way would require a significant effort. So
3518 much so that it makes me question the applicability of structural differencing for the
3519 exclusive purpose of merging source-code. From a broader perspective, however, there
3520 are many other interesting applications that could benefit from structural differencing
3521 techniques. In particular, we can probably use structural differencing to aid any task
3522 where a human does not directly edit the files being analyzed or when the result of the
3523 analysis does require no further interaction. For example, it should be possible to deploy
3524 `hdiff` to provide a human readable summary of a patch, something that looks at the
3525 working directory, computes the structural diffs between the various files, just like `git`
3526 `diff`, but displays information in the lines of:

```
3527 some/project/dir $ hsummary
3528 function fact refactored;
3529 definition of fact changed;
3530 import statements added;
```

3531 In combination with the powerful web interfaces of services like GitHub or GitLab,
3532 we could also use tools like `hdiff` to study the evolution of code or to inform the assignee
3533 of a pull request whether or not it detected the changes to be *structurally disjoint*. If
3534 nothing else, we could at least direct the attention of the developers to the locations
3535 in the source-code where there are actual conflicts and the developer has to make a
3536 choice. That is where mistakes are more likely to be made. One way of circumventing the
3537 formatting and comment issues above is to write a tool that checks whether the developer
3538 included all changes in a sensible way and warns them otherwise, but it is always a
3539 human performing the actual merge.

3540 Finally, differencing file formats that are based on JSON or XML, such as document
3541 processors and spreadsheet processors, might be much easier than source code. Take the
3542 formatting of a `.odf` file for example. It is automatically generated and independent of
3543 the formatting of document inside the file and it has no scoping or sharing inside, hence,
3544 it would be simpler to deploy a structural merging tool over `.odf` files. Some care must
3545 be taken with the unordered trees, even though I conjecture `hdiff` would behave mostly
3546 alright.

3547 7.2 CONCLUDING REMARKS

3548 This dissertation explored a novel approach to structural differencing and a success-
3549 ful prototype for computing and merging patches for said approach. The main novelty
3550 comes from relying on unrestricted tree-matchings, which are possible because we never
3551 translate to an edit-script-like structure. We have identified the challenges of employing
3552 such techniques to merging of source-code but still achieved encouraging empirical re-
3553 sults. In the process of developing our prototypes we have also improved the Haskell
3554 ecosystem for generic programming.

3555



3556

3557 **SOURCE-CODE AND DATASET**

3558 **A.1 SOURCE-CODE**

3559 The easiest way to obtain the source is through either GitHub or Hackage. The source
3560 code for the different projects discussed throughout this dissertation is publicly available
3561 as Haskell packages, on Hackage:

- 3562 • `hackage.haskell.org/package/generics-mrsop`
- 3563 • `hackage.haskell.org/package/simplistic-generics`
- 3564 • `hackage.haskell.org/package/generics-mrsop-gdiff`
- 3565 • `hackage.haskell.org/package/hdiff`

3566 The actual version of `hdiff` that we have documented and used to obtain the results
3567 presented in this dissertation has been archived on Zenodo [68].

3568 **A.2 DATASET**

3569 The dataset [67] was obtained by running the data collection script (Section 6.1) over
3570 the repositories listed in Table A.1, on the 16th of January of 2020. It is also available in
3571 Zenodo for download.

TABLE A.1: *Repositories used for data collection*

Language	Repository	Conflicts	Commits	Forks
Clojure	metabase/metabase	411	18697	25
Clojure	onyx-platform/onyx	189	6828	209
Clojure	incanter/incanter	96	1593	286
Clojure	nathanmarz/cascalog	68	1366	181
Clojure	overtone/overtone	65	3070	413
Clojure	technomancy/leiningen	46	4736	15
Clojure	ring-clojure/ring	44	1027	441
Clojure	ztellman/aleph	43	1398	213
Clojure	pedestal/pedestal	35	1581	248
Clojure	circleci/frontend	33	18857	170
Clojure	arcadia-unity/Arcadia	25	1716	95
Clojure	walmartlabs/lacinia	19	991	105
Clojure	clojure/clojurescript	18	5706	730
Clojure	oakes/Nightcode	17	1914	119
Clojure	weavejester/compojure	16	943	245
Clojure	boot-clj/boot	12	1331	169
Clojure	clojure-liberator/liberator	12	406	144
Clojure	originrose/cortex	11	1045	103
Clojure	dakrone/clj-http	9	1198	368
Clojure	bhauman/lein-figwheel	9	1833	221
Clojure	jonase/kibit	9	436	124
Clojure	riemann/riemann	7	1717	512
Clojure	korma/Korma	7	491	232
Clojure	clojure/core.async	4	564	181
Clojure	status-im/status-react	3	5224	723
Clojure	cemerick/friend	2	227	122
Clojure	LightTable/LightTable	1	1265	927
Clojure	krisajenkins/yesql	1	285	112
Clojure	cgrand/enlive	1	321	144
Clojure	plumatic/schema	1	825	244
Java	spring-projects/spring-boot	760	24545	284
Java	elastic/elasticsearch	746	49920	158
Java	apereo/cas	363	15834	31
Java	jenkinsci/jenkins	296	29141	6
Java	xetorthio/jedis	147	1610	32
Java	google/ExoPlayer	133	7694	44
Java	apache/storm	117	10204	4
Java	junit-team/junit4	77	2427	29
Java	skylot/jadx	52	1165	24
Java	naver/pinpoint	51	10931	3
Java	apache/beam	34	25062	22
Java	baomidou/mybatis-plus	31	3640	21
Java	mybatis/mybatis-3	21	3164	83
Java	dropwizard/dropwizard	20	5229	31
Java	SeleniumHQ/selenium	18	24627	54

TABLE A.1: *Repositories used for data collection (continued)*

Language	Repository	Conflicts	Commits	Forks
Java	codecraft/webmagic	11	1015	37
Java	aws/aws-sdk-java	7	2340	24
Java	spring-projects/spring-security	7	8339	36
Java	eclipse/deeplearning4j	6	572	48
Java	square/okhttp	5	4407	78
JavaScript	meteor/meteor	1208	22501	51
JavaScript	adobe/brackets	699	17782	66
JavaScript	mrdoob/three.js	403	31473	22
JavaScript	moment/moment	141	3724	65
JavaScript	RocketChat/Rocket.Chat	125	17445	55
JavaScript	serverless/serverless	118	12278	39
JavaScript	nodejs/node	99	29302	159
JavaScript	twbs/bootstrap	86	19261	679
JavaScript	photonstorm/phaser	80	13958	61
JavaScript	emberjs/ember.js	76	19460	42
JavaScript	atom/atom	63	37335	137
JavaScript	TryGhost/Ghost	50	10374	7
JavaScript	jquery/jquery	44	6453	19
JavaScript	mozilla/pdf.js	41	12132	69
JavaScript	Leaflet/Leaflet	37	6810	44
JavaScript	expressjs/express	36	5558	79
JavaScript	hexojs/hexo	27	3146	38
JavaScript	videojs/video.js	17	3509	63
JavaScript	facebook/react	10	12732	273
JavaScript	jashkenas/underscore	8	2447	55
JavaScript	lodash/lodash	8	7992	46
JavaScript	axios/axios	8	900	6
JavaScript	select2/select2	3	2573	58
JavaScript	chartjs/Chart.js	3	2966	101
JavaScript	facebook/jest	2	4595	41
JavaScript	vuejs/vue	1	3076	234
JavaScript	nwjs/nw.js	1	3913	38
Lua	Kong/kong	209	5494	31
Lua	hawkthorne/hawkthorne-journey	155	5538	370
Lua	snabbco/snabb	119	9456	295
Lua	tarantool/tarantool	54	13542	224
Lua	luarocks/luarocks	45	2325	296
Lua	luakit/luakit	28	4186	219
Lua	pkulchenko/ZeroBraneStudio	20	3945	447
Lua	CorsixTH/CorsixTH	16	3355	250
Lua	OpenNMT/OpenNMT	14	1684	455
Lua	koreader/koreader	14	7256	710
Lua	bakpakin/Fennel	12	689	59
Lua	Olivine-Labs/busted	9	950	139

TABLE A.1: *Repositories used for data collection (continued)*

Language	Repository	Conflicts	Commits	Forks
Lua	Element-Research/rnn	8	622	318
Lua	lcpz/awesome-copypcats	8	821	412
Lua	Tieske/Penlight	6	743	190
Lua	yagop/telegram-bot	5	729	519
Lua	awesomeWM/awesome	5	9990	360
Lua	torch/nn	4	1839	967
Lua	luvit/luvit	4	2897	330
Lua	GUI/lua-resty-auto-ssl	3	318	119
Lua	alexazhou/VeryNginx	3	604	810
Lua	sailorproject/sailor	2	640	128
Lua	leafo/moonscript	2	738	162
Lua	nrk/redis-lua	1	327	193
Lua	skywind3000/z.lua	1	367	59
Lua	rxl/json.lua	1	46	144
Lua	luafun/luafun	1	55	88
Python	python/cpython	891	106167	131
Python	sympy/sympy	864	41009	29
Python	matplotlib/matplotlib	515	32949	47
Python	home-assistant/home-assistant	496	23812	91
Python	bokeh/bokeh	326	18196	32
Python	certbot/certbot	272	9524	28
Python	scikit-learn/scikit-learn	192	25044	19
Python	explosion/spaCy	163	11141	27
Python	docker/compose	129	5590	29
Python	scrapy/scrapy	74	7705	83
Python	keras-team/keras	70	5342	176
Python	tornadoweb/tornado	60	4144	51
Python	pallets/flask	56	3799	132
Python	ipython/ipython	51	24203	39
Python	pandas-dev/pandas	48	21596	92
Python	quantopian/zipline	45	6032	31
Python	Theano/Theano	44	28099	25
Python	psf/requests	32	5927	75
Python	ansible/ansible	29	48864	18
Python	nvbn/thefuck	11	1555	26
Python	waditu/tushare	8	407	35
Python	facebook/prophet	4	445	26
Python	jakubroztocil/httpie	3	1145	29
Python	binux/pyspider	1	1174	34
Python	Jack-Cherish/python-spider	1	279	39
Python	zulip/zulip	1	34149	35

BIBLIOGRAPHY

- 3574 [1] ADAMS, M. D. Scrap Your Zippers: A Generic Zipper for Heterogeneous Types. In
3575 *WGP '10: Proceedings of the 2010 ACM SIGPLAN workshop on Generic program-*
3576 *ming* (New York, NY, USA, 2010), ACM, pp. 13–24.
- 3577 [2] AKUTSU, T. Tree edit distance problems: Algorithms and applications to bioinfor-
3578 matics. *IEICE Transactions on Information and Systems E93.D*, 2 (2010), 208–218.
- 3579 [3] AKUTSU, T., FUKAGAWA, D., AND TAKASU, A. Approximating tree edit distance
3580 through string edit distance. *Algorithmica* 57, 2 (Jun 2010), 325–348.
- 3581 [4] ALTENKIRCH, T., GHANI, N., HANCOCK, P., MCBRIDE, C., AND MORRIS, P. In-
3582 dexed containers. *Journal of Functional Programming* 25 (2015).
- 3583 [5] ANDERSEN, J., AND LAWALL, J. L. Generic patch inference. In *23rd IEEE/ACM In-*
3584 *ternational Conference on Automated Software Engineering* (L'Aquila, Italy, Sept.
3585 2008), pp. 337–346.
- 3586 [6] ASENOV, D., GUENAT, B., MÜLLER, P., AND OTTH, M. Precise version control of
3587 trees with line-based version control systems. In *Proceedings of the 20th Interna-*
3588 *tional Conference on Fundamental Approaches to Software Engineering - Volume*
3589 *10202* (New York, NY, USA, 2017), Springer-Verlag New York, Inc., pp. 152–169.
- 3590 [7] AUGSTEN, N., BOHLEN, M., DYRESON, C., AND GAMPER, J. Approximate joins
3591 for data-centric xml. In *2008 IEEE 24th International Conference on Data Engi-*
3592 *neering* (2008), IEEE, pp. 814–823.
- 3593 [8] AUGSTEN, N., BÖHLEN, M., AND GAMPER, J. The pq-gram distance between or-
3594 dered labeled trees. *ACM Transactions on Database Systems (TODS)* 35, 1 (2010),
3595 4.
- 3596 [9] AUTEXIER, S. Similarity-based diff, three-way diff and merge. *Int. J. Software and*
3597 *Informatics* 9, 2 (2015), 259–277.
- 3598 [10] BACKURS, A., AND INDYK, P. Edit distance cannot be computed in strongly sub-
3599 quadratic time (unless seth is false). In *Proceedings of the Forty-Seventh Annual*
3600 *ACM Symposium on Theory of Computing* (New York, NY, USA, 2015), STOC '15,
3601 Association for Computing Machinery, p. 51–58.
- 3602 [11] BALASUBRAMANIAM, S., AND PIERCE, B. C. What is a file synchronizer? In *Pro-*
3603 *ceedings of the 4th Annual ACM/IEEE International Conference on Mobile Com-*
3604 *puting and Networking* (New York, NY, USA, 1998), MobiCom '98, ACM, pp. 98–
3605 108.

- [12] BARENDREGT, H. P. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier, 1984.
- [13] BERGROTH, L., HAKONEN, H., AND RAITA, T. A survey of longest common subsequence algorithms. In *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)* (Washington, DC, USA, 2000), SPIRE '00, IEEE Computer Society, pp. 39–.
- [14] BEZEM, M., KLOP, J., BARENDSEN, E., DE VRIJER, R., AND TERESE. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.
- [15] BILLE, P. A survey on tree edit distance and related problems. *Theor. Comput. Sci.* 337, 1-3 (June 2005), 217–239.
- [16] BOTTU, G.-J., KARACHALIAS, G., SCHRIJVERS, T., OLIVEIRA, B. C. D. S., AND WADLER, P. Quantified class constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell* (New York, NY, USA, 2017), Haskell 2017, ACM, pp. 148–161.
- [17] BRAM COHEN. Patience Diff Advantages, 2010. <https://bramcohen.livejournal.com/73318.html>.
- [18] BRASS, P. *Advanced Data Structures*. Cambridge University Press, 2008.
- [19] BRAVENBOER, M., KALLEBERG, K. T., VERMAAS, R., AND VISSER, E. Stratego/xt 0.17. a language and toolset for program transformation. *Science of computer programming* 72, 1-2 (2008), 52–70.
- [20] CHAWATHE, S. S., AND GARCIA-MOLINA, H. Meaningful change detection in structured data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1997), SIGMOD '97, ACM, pp. 26–37.
- [21] CHAWATHE, S. S., RAJARAMAN, A., GARCIA-MOLINA, H., AND WIDOM, J. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1996), SIGMOD '96, ACM, pp. 493–504.
- [22] COBÉNA, G., ABITEBOUL, S., AND MARIAN, A. Detecting changes in xml documents. pp. 41–52.
- [23] DE VRIES, E., AND LÖH, A. True Sums of Products. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming* (New York, NY, USA, 2014), WGP '14, ACM, pp. 83–94.

- [24] DEMAINE, E. D., MOZES, S., ROSSMAN, B., AND WEIMANN, O. An optimal decomposition algorithm for tree edit distance. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP 2007)* (Wroclaw, Poland, July 9–13 2007), pp. 146–157.
- [25] DULUCQ, S., AND TOUZET, H. Analysis of tree edit distance algorithms. In *Combinatorial Pattern Matching* (Berlin, Heidelberg, 2003), R. Baeza-Yates, E. Chávez, and M. Crochemore, Eds., Springer Berlin Heidelberg, pp. 83–95.
- [26] EISENBERG, R. A., AND WEIRICH, S. Dependently typed programming with singletons. In *Proceedings of the 2012 Haskell Symposium* (New York, NY, USA, 2012), Haskell '12, ACM, pp. 117–130.
- [27] EISENBERG, R. A., WEIRICH, S., AND AHMED, H. G. Visible Type Application. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings* (2016), pp. 229–254.
- [28] FALLERI, J., MORANDAT, F., BLANC, X., MARTINEZ, M., AND MONPERRUS, M. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014* (2014), pp. 313–324.
- [29] FARINIER, B., GAZAGNAIRE, T., AND MADHAVAPEDDY, A. Mergeable persistent data structures. In *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)* (Le Val d'Ajol, France, Jan. 2015), D. Baelde and J. Alglave, Eds.
- [30] FILLIÂTRE, J.-C., AND CONCHON, S. Type-safe modular hash-consing. In *Proceedings of the 2006 Workshop on ML* (New York, NY, USA, 2006), ML '06, ACM, pp. 12–19.
- [31] FINIS, J. P., RAIBER, M., AUGSTEN, N., BRUNEL, R., KEMPER, A., AND FÄRBER, F. Rws-diff: Flexible and efficient change detection in hierarchical data. In *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management* (New York, NY, USA, 2013), CIKM '13, ACM, pp. 339–348.
- [32] FOSTER, J. N., GREENWALD, M. B., KIRKEGAARD, C., PIERCE, B. C., AND SCHMITT, A. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences* 73, 4 (2007), 669–689.
- [33] GARUFFI, G. Version control systems: Diffing with structure. Master's thesis, Utrecht Universiteit, 2018.
- [34] GHANI, N., LÜTH, C., DE MARCHI, F., AND POWER, J. Algebras, coalgebras, monads and comonads. *Electronic Notes in Theoretical Computer Science* 44, 1 (2001), 128–145.

- [35] GIBBONS, J. Design patterns as higher-order datatype-generic programs. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Generic Programming* (New York, NY, USA, 2006), WGP '06, ACM, pp. 1–12.
- [36] GOTO, E. Monocopy and associative algorithms in an extended lisp. Tech. rep., University of Tokyo, 1974.
- [37] GUHA, S., JAGADISH, H., KOUDAS, N., SRIVASTAVA, D., AND YU, T. Approximate xml joins. pp. 287–298.
- [38] HASHIMOTO, M., AND MORI, A. Diff/ts: A tool for fine-grained structural change analysis. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering* (Washington, DC, USA, 2008), WCRE '08, IEEE Computer Society, pp. 279–288.
- [39] HENIKOFF, S., AND HENIKOFF, J. G. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences* 89, 22 (1992), 10915–10919.
- [40] HINZE, R., JEURING, J., AND LÖH, A. Type-indexed data types. *Science of Computer Programming* 51, 1 (2004), 117 – 151. Mathematics of Program Construction (MPC 2002).
- [41] HUET, G. The zipper. *J. Funct. Program.* 7 (09 1997), 549–554.
- [42] HUNT, J. W., AND MCILROY, M. D. An algorithm for differential file comparison. Tech. Rep. CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.
- [43] JONES, S. P., WEIRICH, S., EISENBERG, R. A., AND VYTINIOTIS, D. A reflection on types. In *A List of Successes That Can Change the World*. Springer, 2016, pp. 292–317.
- [44] KHANNA, S., KUNAL, K., AND PIERCE, B. C. A formal investigation of diff3. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science* (Berlin, Heidelberg, 2007), V. Arvind and S. Prasad, Eds., Springer Berlin Heidelberg, pp. 485–496.
- [45] KIM, D., NAM, J., SONG, J., AND KIM, S. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE Press, pp. 802–811.
- [46] KLEIN, P. N. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th Annual European Symposium on Algorithms* (London, UK, UK, 1998), ESA '98, Springer-Verlag, pp. 91–102.

- [47] KLINT, P., VAN DER STORM, T., AND VINJU, J. Rascal: A domain specific language for source code analysis and manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation* (2009), IEEE, pp. 168–177.
- [48] KNUTH, D. E. The genesis of attribute grammars. In *Proceedings of the International Conference on Attribute Grammars and Their Applications* (Berlin, Heidelberg, 1990), WAGA, Springer-Verlag, p. 1–12.
- [49] LÄMMEL, R., AND JONES, S. P. 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* (New York, NY, USA, 2003), TLDI '03, ACM, pp. 26–37.
- [50] LANHAM, M., KANG, A., HAMMER, J., HELAL, A., AND WILSON, J. Format-independent change detection and propagation in support of mobile computing. *Brazilian Symposium on Databases (SBB)* (01 2002), 27–41.
- [51] LEMPSINK, E., LEATHER, S., AND LÖH, A. Type-safe diff for families of datatypes. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming* (New York, NY, USA, 2009), WGP '09, ACM, pp. 61–72.
- [52] LEVENSHEIN, V. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (1966), 707.
- [53] LINDHOLM, T. A three-way merge for xml documents. In *Proceedings of the 2004 ACM Symposium on Document Engineering* (New York, NY, USA, 2004), DocEng '04, ACM, pp. 1–10.
- [54] LÖH, A., AND MAGALHAES, J. P. Generic programming with indexed functors. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming* (2011), ACM, pp. 1–12.
- [55] MAGALHÃES, J. P., DIJKSTRA, A., JEURING, J., AND LÖH, A. A Generic Deriving Mechanism for Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell* (New York, NY, USA, 2010), Haskell '10, ACM, pp. 37–48.
- [56] MAGALHÃES, J. P., AND LÖH, A. A Formal Comparison of Approaches to Datatype-Generic Programming. In *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, Tallinn, Estonia, 25 March 2012* (2012), J. Chapman and P. B. Levy, Eds., vol. 76 of *Electronic Proceedings in Theoretical Computer Science*, Open Publishing Association, pp. 50–67.
- [57] MALETIC, J. I., AND COLLARD, M. L. Exploration, analysis, and manipulation of source code using srcml. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2* (2015), ICSE '15, IEEE Press, p. 951–952.

- [58] MARLOW, S., ET AL. Haskell 2010 Language Report, 2010. <https://www.haskell.org/onlinereport/haskell2010/>.
- [59] MCBRIDE, C. The derivative of a regular type is its type of one-hole contexts (extended abstract), 2001.
- [60] MCBRIDE, C. *Epigram: Practical Programming with Dependent Types*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 130–170.
- [61] MCBRIDE, C., AND MCKINNA, J. The view from the left. *J. Funct. Program.* 14, 1 (Jan. 2004), 69–111.
- [62] MCKENNA, A., HANNA, M., BANKS, E., SIVACHENKO, A., CIBULSKIS, K., KERNYTSKY, A., GARIMELLA, K., ALTSHULER, D., GABRIEL, S., DALY, M., AND DEPRISTO, M. The genome analysis toolkit: A mapreduce framework for analyzing next-generation dna sequencing data. *Genome research* 20 (09 2010), 1297–303.
- [63] MEDEIROS, F., RIBEIRO, M., GHEYI, R., APEL, S., KÄSTNER, C., FERREIRA, B., CARVALHO, L., AND FONSECA, B. Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. *IEEE Transactions on Software Engineering* 44, 5 (2017), 453–469.
- [64] MENEZES A. J., P. V. O., AND VANSTONE, S. A. *Handbook of Applied Cryptography*, boca raton, xiii, 780, 1997 ed. CRC Press.
- [65] MERKLE, R. C. A digital signature based on a conventional encryption function. In *Advances in Cryptology — CRYPTO '87* (Berlin, Heidelberg, 1988), C. Pomerance, Ed., Springer Berlin Heidelberg, pp. 369–378.
- [66] MIMRAM, S., AND GIUSTO, C. D. A categorical theory of patches. *CoRR abs/1311.3903* (2013).
- [67] MIRALDO, V. C. Dataset of merge conflicts collected from GitHub repositories. <https://doi.org/10.5281/zenodo.3751038>, Apr. 2020.
- [68] MIRALDO, V. C. hdiff: hash-based differencing of structured data. <https://doi.org/10.5281/zenodo.3754632>, Apr. 2020.
- [69] MIRALDO, V. C., DAGAND, P.-E., AND SWIERSTRA, W. Type-directed diffing of structured data. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Type-Driven Development* (New York, NY, USA, 2017), TyDe 2017, ACM, pp. 2–15.
- [70] MIRALDO, V. C., AND SERRANO, A. 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* (2018), ACM, pp. 65–77.

- 3782 [71] MIRALDO, V. C., AND SWIERSTRA, W. An efficient algorithm for type-safe struc-
3783 tural diffing. *PACMPL* 3, ICFP (2019), 113:1–113:29.
- 3784 [72] MITCHELL, N., AND RUNCIMAN, C. Uniform Boilerplate and List Processing. In
3785 *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop* (New York, NY,
3786 USA, 2007), Haskell '07, ACM, pp. 49–60.
- 3787 [73] MOUAT, A. Xml diff and patch utilities. Master's thesis, Heriot-Watt University,
3788 Edinburgh, 2002.
- 3789 [74] NAVARRO, G. A guided tour to approximate string matching. *ACM Comput. Surv.*
3790 33, 1 (Mar. 2001), 31–88.
- 3791 [75] NERON, P., TOLMACH, A., VISSER, E., AND WACHSMUTH, G. A theory of name
3792 resolution. vol. 9032.
- 3793 [76] NOORT, T. V., RODRIGUEZ, A., HOLDERMANS, S., JEURING, J., AND HEEREN, B. A
3794 Lightweight Approach to Datatype-generic Rewriting. In *Proceedings of the ACM*
3795 *SIGPLAN Workshop on Generic Programming* (New York, NY, USA, 2008), WGP
3796 '08, ACM, pp. 13–24.
- 3797 [77] NORELL, U. Dependently typed programming in agda. In *International school on*
3798 *advanced functional programming* (2008), Springer, pp. 230–266.
- 3799 [78] OKASAKI, C., AND GILL, A. Fast mergeable integer maps. In *In Workshop on ML*
3800 (1998), pp. 77–86.
- 3801 [79] PAAßEN, B. Revisiting the tree edit distance and its backtracing: A tutorial. *CoRR*
3802 *abs/1805.06869* (2018).
- 3803 [80] PAAßEN, B., HAMMER, B., PRICE, T. W., BARNES, T., GROSS, S., AND PINKWART,
3804 N. The continuous hint factory - providing hints in vast and sparsely populated
3805 edit distance spaces. *CoRR abs/1708.06564* (2017).
- 3806 [81] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., LAWALL, J., AND MULLER, G.
3807 Faults in Linux: Ten years later. In *Sixteenth International Conference on Architec-*
3808 *tural Support for Programming Languages and Operating Systems (ASPLOS 2011)*
3809 (Newport Beach, CA, USA, Mar. 2011).
- 3810 [82] PAWLIK, M., AND AUGSTEN, N. RTED: A robust algorithm for the tree edit dis-
3811 tance. *CoRR abs/1201.0230* (2012).
- 3812 [83] PETERS, L. Change detection in xml trees: a survey. In *3rd Twente Student Con-*
3813 *ference on IT* (2005).
- 3814 [84] PICKERING, M., ÉRDI, G., PEYTON JONES, S., AND EISENBERG, R. A. Pattern
3815 synonyms. In *Proceedings of the 9th International Symposium on Haskell* (New
3816 York, NY, USA, 2016), Haskell 2016, ACM, pp. 80–91.

- 3817 [85] PLOTKIN, G. Further note on inductive generalization. *Machine Intelligence*. 6
3818 (01 1971).
- 3819 [86] PUTTEN, A. V. Generic diffing and merging of mutually recursive datatypes in
3820 haskell. Master's thesis, Utrecht Universiteit, 2019.
- 3821 [87] RANDY SMITH. Version 3.7, December 2018; distributed with GNU diffutils
3822 package, 1988. <http://ftp.gnu.org/gnu/diffutils/>.
- 3823 [88] ROBINSON, E., AND ROSOLINI, G. Categories of partial maps. *Information and*
3824 *computation* 79, 2 (1988), 95–130.
- 3825 [89] ROBINSON, J. A. A machine-oriented logic based on the resolution principle. *J.*
3826 *ACM* 12, 1 (Jan. 1965), 23–41.
- 3827 [90] RODRIGUEZ, A., JEURING, J., JANSSON, P., GERDES, A., KISELYOV, O., AND
3828 OLIVEIRA, B. C. D. S. Comparing Libraries for Generic Programming in Haskell.
3829 In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (New York, NY,
3830 USA, 2008), Haskell '08, ACM, pp. 111–122.
- 3831 [91] ROY CHOUDHURY, R., YIN, H., AND FOX, A. Scale-driven automatic hint genera-
3832 tion for coding style. In *Intelligent Tutoring Systems* (Cham, 2016), A. Micarelli,
3833 J. Stamper, and K. Panourgia, Eds., Springer International Publishing, pp. 122–
3834 132.
- 3835 [92] SERRANO, A., AND HAGE, J. Generic matching of tree regular expressions over
3836 haskell data types. In *Practical Aspects of Declarative Languages - 18th Interna-*
3837 *tional Symposium, PADL 2016, St. Petersburg, FL, USA, January 18-19, 2016. Pro-*
3838 *ceedings* (2016), pp. 83–98.
- 3839 [93] SERRANO, A., AND MIRALDO, V. C. Generic programming of all kinds. In *Pro-*
3840 *ceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (2018),
3841 ACM, pp. 41–54.
- 3842 [94] SHAPIRA, D., AND STORER, J. A. Edit distance with move operations. In *Combina-*
3843 *torial Pattern Matching* (Berlin, Heidelberg, 2002), A. Apostolico and M. Takeda,
3844 Eds., Springer Berlin Heidelberg, pp. 85–98.
- 3845 [95] SHEARD, T., AND JONES, S. P. Template meta-programming for haskell. In *Pro-*
3846 *ceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (New York, NY, USA,
3847 2002), Haskell '02, ACM, pp. 1–16.
- 3848 [96] THIJE, J., AND ZEEVAERT, L. *Receptive Multilingualism: Linguistic Analyses, Lan-*
3849 *guage Policies, and Didactic Concepts*. Hamburg studies on multilingualism. J.
3850 Benjamins Publishing Company, 2007.

- [97] TILMANN, F. latexdiff, 2004. mirrors.ctan.org/support/latexdiff/doc/latexdiff-man.pdf.
- [98] VAN HOREBEEK, I., AND LEWI, J. *Abstract Data Types as Initial Algebras*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1989, pp. 14–65.
- [99] VAN TONDER, R., AND LE GOUES, C. Lightweight multi-language syntax transformation with parser parser combinators. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019), ACM, pp. 363–378.
- [100] VASSENA, M. Generic diff3 for algebraic datatypes. In *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016* (2016), pp. 62–71.
- [101] WADLER, P. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1987), POPL ’87, pp. 307–313.
- [102] WEIRICH, S., HSU, J., AND EISENBERG, R. A. System FC with Explicit Kind Equality. *SIGPLAN Not.* 48, 9 (Sept. 2013), 275–286.
- [103] WEIRICH, S., VOIZARD, A., DE AMORIM, P. H. A., AND EISENBERG, R. A. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP (Aug. 2017), 31:1–31:29.
- [104] XI, H., CHEN, C., AND CHEN, G. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2003), POPL ’03, ACM, pp. 224–235.
- [105] YAKUSHEV, A. R., HOLDERMANS, S., LÖH, A., AND JEURING, J. Generic Programming with Fixed Points for Mutually Recursive Datatypes. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2009), ICFP ’09, ACM, pp. 233–244.
- [106] YORGEY, B. A., WEIRICH, S., CRETIN, J., PEYTON JONES, S., VYTINIOTIS, D., AND MAGALHÃES, J. P. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (New York, NY, USA, 2012), TLDI ’12, ACM, pp. 53–66.
- [107] ZHANG, K., AND SHASHA, D. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing* 18, 6 (1989), 1245–1262.
- [108] ZHANG, K., STATMAN, R., AND SHASHA, D. On the editing distance between unordered labeled trees. *Information processing letters* 42, 3 (1992), 133–139.