UTRECHT UNIVERSITY

MASTER THESIS

SVC
A prototype of a Structure-aware Version Control system

Author:
Marco VASSENA
ICA-4110161

Supervisor:
Dr. Wouter SWIERSTRA
Prof. Johan JEURING

Software Technology Group
Computing Science

August 31, 2015

Abstract

This thesis studies the problem of structure-aware revision control, which consists of exploiting the knowledge of the structure of data to improve the quality of version control systems. Formats are firstly described using an EDSL, which distinguishes meta-data from the actual content. From the unique format specification inverse-by-construction parser and printer are derived. The data stored in a file is converted into a heterogeneous rose tree, a generic representation of algebraic data types, used by a `diff` and `diff3` algorithm to respectively detect changes and merge revisions. Lastly the semantics and the properties of the two algorithms are studied with a formal model developed in the Agda proof assistant.

Contents

Chapter 1

Introduction

The amount of data produced has grown in the past few decades. Nowadays data is stored in a variety of different formats and shared on the cloud. In collaborative environments, such as the wiki web application, it is accessible at the same time from multiple devices and can be modified, often concurrently, by the users. In these settings it is crucial to handle changes to data appropriately. For example it is important to keep track of the history of each object, so that at any time it is possible to roll back to a previous version and it should strive to automatically merge simultaneous changes as much as possible.

1.1 Description of the problem

In software industry specific tools called Version Control Systems (VCS), such as Git [11] and Mercurial [48], address the problem of content management for software artifacts. Those tools employ line-based diff algorithms, such as GNU diff, that detect changes line by line and therefore are suitable for source code. However they are not appropriate for any kind of data, because some are not naturally organized in lines of text. For example binary formats, such as videos and images, are not supported and even text-based formats, such as HTML and XML, are poorly supported, because single lines do not reflect the underlying tree-structure of their content.

Whenever revision control is needed, embedded in a specific domain, these systems must be reimplemented, tailored on the specifics of the formats involved. In particular the key components of these tools are a comparison function `diff`, which detects and collects the differences between two objects in an edit script, and its inverse function `patch`, which applies the edits stored in a script.

In revision control simultaneous changes need to be merged. Sometimes this can be achieved automatically by merging algorithms such as the (recursive) *three-way merge* and patch commutation [57], otherwise a conflict is detected and the user must manually solve it, combining the conflicting changes appropriately. The GNU diff3 tool is a line-based algorithm employed in the *three-way merge*, that compares three files, the original version and two modified versions of that file. When it succeeds it creates an edit script that includes changes from both the new versions and whose patch, when applied to the original file, produces a file that merges both. However a line-based approach restricts the opportunities of automatic merging for non line-structured data, resulting in a greater number of conflicts and ultimately additional burden to the user.

Furthermore, even though revision systems are widely spread, their behaviour is understood mostly empirically. As a result the outcome of complex merges is often hard to predict and source of bug reports.

1.2 Research Questions

This master thesis aims to develop a prototype of a structure-aware version control system that addresses the shortcomings of widespread industrial-strength version control systems, which employs line-based algorithms. In particular the prototype exploits the knowledge of data formats, in order to compute more precise diffs that take into account the structure of the content of files, hence improving the quality of revision control. Specifically such a system avoids unnecessary conflicts and increases its automatic merging capabilities.

In order to develop such a system, the thesis addresses the following specific problems.

Data Format Description A structure-aware version control system needs to retrieve the actual data from each file, in order to precisely diff the content, rather than its representation on disk, consequently each file has to parsed with respect to its format. Furthermore after merging the data stored in two versions of the same file, it has to serialize it back to disk for persistency. In this process it is essential to ensure consistency between the parsing and unparsing function.

Is is possible to derive both parser and printer from a format description?

Generic Diff and Diff$_3$ Considering the profusion and the complexity of data formats, it is impractical to study specific diffing and merging algorithms for each of them. However the data retrieved by parsers is always structured in a parse tree, which is usually represented as a domain-specific algebraic data type.

Is it possible to exploit generic programming techniques to implement a generic diff and merge algorithm for structured data?

Formal Model Nowadays version control system play a vital role in collaborative environments, therefore their semantics ought to be put on a formal footing. Their core features, management and merge of revisions, rely on diffing and merging algorithms, which should then be studied closely.

How can we formalize diffing and merging algorithms and study their properties?

1.3 Overview

Chapter 2 presents an embedded domain specific language for formats. The EDSL allows to describe binary and text data formats, from which it is possible to derive automatically inverse parsers and printers. The implementation is tested with real-world formats such as portable bit map images (PBM), portable grey image format (PGM) and comma separated values (CSV), HTML and XML.

Chapter 3 presents a formal model, developed in the Agda proof assistant [47, 9], used to study the diff and merge algorithms. The model is used to show that the two algorithms satisfy their specifications and to prove a number of properties. In particular necessary and sufficient conditions for the presence of conflicts are given.

In chapter 4 the algorithms studied in the model are implemented in the Haskell programming language [40]. Exploiting advanced type features a sizable amount of type-safety is retained in the translation from the formal model, developed in a language with fully-fledged dependent types, to Haskell. The chapter includes also a proof-of-concept structure-aware version control system that employs the ideas discussed in this thesis.

Lastly chapter 6 concludes, summarizing the contributions of this thesis and presenting ideas for future work.

Chapter 2

Format Representation

This chapter addresses the problem of finding a unique representation for data formats, which is an important component of a structure-aware version control system. Section 2.1 introduces the problem by analyzing the issue in more detail 2.1.1 and briefly covers several auxiliary concepts, such as heterogeneous lists 2.1.2 and partial isomorphism 2.1.3. The basic format representation is presented in section 2.2, then several extensions are discussed in 2.3. Section 2.4 concludes, summarizing the main contributions of this chapter 2.4.1 and reviewing related work 2.4.2.

2.1 Introduction

2.1.1 Motivation

A format specifies how some data is stored and encoded in a file. When the format of a file is known, it is possible to decode it and apply the diff algorithm on the data itself, rather than its representation, thus leading to more accurate edit scripts. These are then applied by a patch algorithm to the decoded data, producing the new version, which is then serialized in a file according to its format specification. A structure-aware version control system exploits this to avoid unnecessary conflicts and thus improving the quality of revision control.

Formats are usually described using formal grammars, such as context free grammars. A parser is a function that recognizes a grammar in the input string and extracts the structured data that it describes. Printing is the opposite function, which consists of serializing some structured data, according to the format specifications. In functional programming languages parser combinators are commonly used to implement parsers. Furthermore there are several tools, such as Yacc [29] and Happy [24], called parser generator, that automatically generate a parser for a given grammar.

Parsers and printers of a format should always be one the inverse of the other in order to ensure round-trip behavior, i.e. serializing some data to file and then parsing it back should yield the initial value. Similarly parsing a properly formatted file and serializing its content should produce the original file. These two functions are usually defined independently with several drawbacks. Firstly it is a repetitive and error-prone task because the two functions share similar information about the format, secondly, as the format changes, the user is burdened with the task of keeping the two functions synchronized. Lastly there is no actual guarantee that the two functions are indeed each other's inverse.

2.1.2 Type List

Haskell type system have been enhanced in GHC with several extensions. The DataKind extension enables user defined kinds, through datatype promotion, i.e. data types are automatically promoted to kinds and their constructors to type constructors. Among these, lists are natively promoted to the kind level, reusing the same syntax. It should always be clear from the context when a list is a value or a type; ambiguous cases will be distinguished prefixing types with a quote. The PolyKinds extension enables kind polymorphism. Kind variables are always universally quantified implicitly, furthermore kinds are inferred automatically, therefore kind annotations are usually superfluous, however few will be left to help the reader.

Examples The constructors for type level lists are kind polymorphic, just like their value counterpart are polymorphic in the type.

```
*> :k '[]
'[] :: [k]
```

Note however that they are homogeneous with respect to the kinds of their elements.

```
*> :k '[Int, Bool]
'[Int, Bool] :: [*]
*> :k '[Maybe, []]
'[Maybe, []] :: [* -> *]
```

In the second example [] is the list type constructor.

Type Family Common functions on lists can be lifted to the type level using a *closed type family*.

For example it is possible to append type level lists just like it happens for concrete lists. The type family contains a recursive call and it is defined by induction on the first list, following closely that of (++) :: [a] -> [a] -> [a].

```
type family (:++:) (xs :: [ k ]) (ys :: [ k ]) :: [ k ] where
  '[] :++: ys = ys
  (x ': xs) :++: ys = x ': (xs :++: ys)
```

Note that just like the standard append is polymorphic in a, so its type level counter part is kind polymorphic. Likewise the input and output lists share the same polymorphic kind k.

```
*> :kind! '[Int, Bool] :++: '[Char, Double]
'[Int, Bool] :++: '[Char, Double] :: [*]
= '[Int, Bool, Char, Double]
```

Analogously a type level map is easily defined:

```
type family Map (f :: k1 -> k2) (xs :: [ k1 ]) :: [ k2 ] where
  Map f '[] = '[]
  Map f (x ': xs) = f x ': Map f xs
```

For example mapping the type constructor [] :: * -> * over a list of types of kind * yields a list of list types.

```
*> :kind! Map [] '[Char, Int , Bool]
Map [] '[Char, Int , Bool] :: [*]
= '[[Char], [Int], [Bool]]
```

Heterogeneous List The data type HList, introduced by Kiselyov [31], is an heterogeneous list, indexed by a type level list. It differs from the conventional homogeneous list data type, because it contain values of possibly different types, that are stored in its index.

```
data HList (xs :: [ * ]) where
  Nil :: HList '[]
  Cons :: x -> HList xs -> HList (x ': xs)
```

The following functions will be assumed in the following. Their implementation is clear from their signatures and therefore omitted. Furthermore they are completely analogous to the correspondent functions for homogeneous lists.

```
hHead :: HList (x ': xs) -> x
hsingleton :: x -> HList '[ x ]
happend :: HList xs -> HList ys -> HList (xs :++: ys)
```

SList *Singleton types* are needed to support dependently typed programming in languages with a strict phase separation between run-time and compile time [20], like Haskell. More precisely a singleton type is a type with only one non-$\perp$ value and represents a run-time witness of a type [20]. The type SList xs is the singleton type for the type level list xs.

```
data SList (xs :: [ * ]) where
  SNil :: SList '[]
  SCons :: SList xs -> SList (x ': xs)
```

Singleton types are essential to implement certain kind of functions, especially when type families are involved. Consider for instance the function split, which splits an heterogeneous list in two parts:

```
split :: HList (xs :++: ys) -> (HList xs, HList ys)
```

The indexes of the two output lists, xs and ys, come from the index of the input list, in which they are appended. Crucially it is not possible to pattern match directly on it, because of the presence of the type family application xs :++: ys in its index: should it be there a case for Nil or Cons ? That depends on the result of xs :++: ys, however it is not possible to pattern match on it directly because it is just a type. Nevertheless it is possible to implement this function if xs is known to be empty or not. A singleton type of type SList xs introduces a true value, which can be inspected providing this piece of information about its index.

```
split :: SList xs -> HList (xs :++: ys) -> (HList xs, HList ys)
split SNil hs = (Nil, hs)
split (SCons s) (Cons h hs) = (Cons h hs1, hs2)
  where (hs1, hs2) = split s hs
```

By pattern matching on it the first part of the list can be distinguished from the second, so that the two can be separated. Specifically when xs is empty the rest of the list belongs to the second part, otherwise the head of the list belongs to the first part.

Often it will be necessary to retrieve a singleton type of a list-indexed data type. The type class Reify is defined for this purpose:

```
class Reify (f :: [ * ] -> *) where
  toSList :: f xs -> SList xs
```

For example it is easy to make HList instance of Reify:

```
instance Reify HList where
  toSList Nil = SNil
  toSList (Cons _ hs) = SCons (toSList hs)
```

2.1.3 Partial Isomorphism

Parsers do not only recognize a grammar in an input string, but usually produce an abstract syntax tree, so that the structured data can be then processed more conveniently. Typically abstract syntax trees are represented directly as a domain specific user-defined data type. On the contrary pretty printing requires to deconstruct the abstract syntax tree, in order to serialize each part appropriately. In order to unify parser and printer in a single entity, constructors and deconstructors of a data type must be coupled together. The partial isomorphism data type is used for this purpose.

```
data Iso xs ys = Iso { apply    :: HList xs -> HList ys,
                       unapply  :: HList ys -> Maybe (HList xs)}
```

A value of type Iso xs ys represents a partial isomorphism, consisting in the mapping apply and its partial inverse unapply. The second function is partial because a data type might have several constructors and therefore the deconstructor of a specific isomorphism could fail if the value has been constructed using another constructor.

Partial isomorphisms form a category:

```
identity :: Iso xs xs
identity = Iso id Just

(.) :: Iso ys zs -> Iso xs ys -> Iso xs zs
(.) g f = Iso s t
  where s = apply g . apply f
        t = unapply g >=> unapply f
```

Example As an example consider the two partial iomorphisms that correspond to list constructors and deconstructors.

```
nil :: Iso '[] '[ [a] ]
nil = Iso f g
```

```
  where f Nil = Cons [] Nil
        g (Cons [] Nil) = Just Nil
        g (Cons (_:_) Nil) = Nothing

cons :: Iso '[a : [a]] '[ [a] ]
cons = Iso f g
  where f (Cons x (Cons xs Nil)) = Cons (x:xs) Nil
        g (Cons [] Nil) = Nothing
        g (Cons (x:xs) Nil) = Just (Cons x (Cons xs Nil))
```

2.2 Format

This thesis addresses the format problem presented in 2.1.1 by implementing an Embedded Domain Specific Language (EDSL) that describes two inverse semantics at once. The format specification is agnostic to the semantics, which is selected afterwards instantiating appropriately a type parameter.

Parsers for context-free grammars are usually implemented using applicative and alternative parser combinators. Since `Functor` is a superclass of `Applicative` and `Applicative` is a superclass of `Alternative`, any instance of the class `Alternative` yields also the `Functor` and `Applicative` instances.

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
pure :: Applicative f => a -> f a
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
(<|>) :: Alternative f => f a -> f a -> f a
empty :: Alternative f => f a
```

Each of this combinator will be encoded as a constructor of the `Format` data type. With an abuse of notation, the operators `<$>`, `<*>`, `<|>`, will be used as the corresponding `Format` constructors. It should always be clear from the context which is which. To keep the description clear, I will firstly discuss formats with token of type `Char` and stream of type `String`, then I will show the changes needed to support any kind of token and stream.

```
data Format (f :: * -> *) (xs :: [ * ]) where
```

A value of type `Format f xs` has underlying semantics `f` and involve values of types `xs`. Furthermore an instance of `Reify (Format f)` is assumed.

The two inverse semantics are represented as an interpretation of this data type.

```
mkParser :: Alternative f => Format f xs -> f (HList xs)
mkPrinter :: Alternative f => Format f xs -> HList xs -> f String
```

2.2.1 Functor Format

The constructor corresponding to the Functor method `fmap` is defined as follows:

```
data Format (f :: * -> *) (xs :: [ * ]) where
  <$> :: Iso xs ys -> Format f xs -> Format f ys
```

The first field is a partial isomorphism that stores a function and its partial inverse.

A parser `f <$> p` applies the function `f` to the result of the parser `p`. If the parser `p` fails, also `f <$> p` fails. This behaviour is accordingly translated in the parsing semantics:

```
mkParser (i <$> f) = apply i <$> mkParser f
```

Note that `apply i` is a pure function.

When printing the format `i <$> f`, the input list is deconstructed using the inverse partial isomoprhism (`unapply`). If it fails the whole printer fails, otherwise the result is used as input to the printer derived from `f`.

```
mkPrinter (i <$> f) hs =
  case unapply i hs of
    Just ys -> mkPrinter f ys
    Nothing -> empty
```

Note that contrary to the standard definition the operator `<$>` is defined right associative:

```
infixr 4 <$>
```

2.2.2 Applicative Format

The constructors corresponding to the Applicative combinators are defined as follows:

```
data Format f xs where
  ...
  Pure :: HList xs -> Format i xs
  (<*>) :: Format f xs -> Format f ys -> Format f (xs :++: ys)
```

The signature of the sequencing operator (<*>) is slightly different from the standard one given in the Applicative class shown previously. This alternative version is actually equivalent, being the star operator ($\star$) from the class Monoidal proposed by McBride and Paterson's [44], revisited to use HList instead of tuples. Also Fokker employs this version in his lecture notes [21]. This symmetric version is more convenient for our purposes therefore it has been preferred over the original one.

The parsing semantics function is standard:

```
mkParser (Pure hs) = pure hs
mkParser (f1 <*> f2) = happend <$> mkParser f1 <*> mkParser f2
```

The format Pure hs is translated to a parser that always returns hs without consuming any input, while the format f1 <*> f2 is converted in a parser that firstly runs the parser of f1 and then that of f2 on the remaining input. The function happend appends the two lists produced by the parsers together, as it is required by the signature of <*>. The parser pure x never fails, while f1 <*> f2 fails if any of the two fails.

Analogously the semantics for the corresponding printer is given by:

```
mkPrinter (Pure _) _ = pure ""
mkPrinter (f1 <*> f2) hs = (++) <$> mkPrinter f1 hs1 <*> mkPrinter f2 hs2
  where (hs1, hs2) = split (toSList f1) hs
```

In the pure case the empty string is returned, because no input is consumed by the corresponding parser. Similarly in the <*> case the two strings produced by the two formats are appended in the same order, to invert the effect of the corresponding parser.

It is straightforward to show that mkPrinter and mkParser represent each other's inverse for the applicative combinators.

2.2.3 Alternative Format

The constructors that correspond to the Alternative combinators have the following signatures:

```
data Format f xs where
  ...
  Empty :: Format i xs
  (<|>) :: Format f xs -> Format f xs -> Format f xs
```

The parser combinator <|> represents choice between parsers. The parser empty always fails without consuming any input and it is the identity of

<|>.

The specific semantics of the choice combinator is most of the time library-dependent and some care is needed to ensure proper invertibility. This issue is discusses in more depth in 2.4.1.

The parsing and printing functions just reuse the Alternative instances of the underlying semantics.

```
mkParser Empty = empty
mkParser (f1 <|> f2) = mkParser f1 <|> mkParser f2

mkPrinter Empty _ = empty
mkPrinter (f1 <|> f2) hs = mkPrinter f1 hs <|> mkPrinter f2 hs
```

2.2.4 Token Format

Parsing libraries usually provide a primitive function to retrieve the next token in the stream, whose type signature is generally similar to the following:

```
pSatisfy :: (Char -> Bool) -> Parser Char
```

If the next token in the stream satisfies the predicate then it is returned, otherwise the parser fails. A corresponding constructor is added to Format:

```
data Format f xs where
  ...
  Satisfy :: (Char -> Bool) -> Format f '[ Char ]
```

The specific primitive used to produce the next token, depends on the on the actual parsing library at hand, therefore a hook is provided in the form of a type class.

```
class ParseSatisfy f where
  satisfy :: (Char -> Bool) -> f Char
```

The parser semantics is given by:

```
mkParser :: (ParseSatisfy f, Alterntaive f) => Format f xs -> f (HList xs)
mkParser (Satisfy p) = hsingleton <$> satisfy p
```

Similarly when printing, a token is produced only if it satisfy the predicate.

```
mkPrinter (Satify p) (Cons c Nil)
  | p c       = pure [ c ]
  | otherwise = empty
```

2.2.5 Example

This section presents few examples that can be already implemented with the Format just defined.

Trivial Format A *trivial format* is a format parametrized by the empty type level list. The parser produced by a trivial format upon success produces the empty HList. The constant partial isomorphism ignore is used to construct trivial formats.

```
ignore :: HList xs -> Iso xs '[]
ignore hs = Iso f g (toSList hs) SNil
  where f _ = Nil
        g _ = Just hs
```

The isomorphism maps any input to the empty list and maps the empty list back to the given list. As a consequence the printer of a trivial format never fails[1].

Essentially a trivial format contains only static information: its parser merely checks the presence of a certain pattern, and its printer just outputs it.

An example of trivial format is the format char c which recognizes a specific character.

```
char :: Char -> Format f '[]
char c = ignore (hsingleton c) <$> satisfy (c ==)
```

The trivial format string s recognizes a specific string and can be obtained by repeatedly applying char.

```
string :: String -> Format f '[]
string [] = unit
string (c:cs) = char c <*> string cs
```

The trivial format unit always succeeds producing the empty HList. It does not consume any input when parsing and that does not output nothing at all when printing.

```
unit :: Format f '[]
unit = Pure Nil
```

Other two useful combinators, included in the Applicative class are:

[1]A case which requires special care is discussed in section 2.2.6

```
(<*) :: f a -> f b -> f a
(*>) :: f a -> f b -> f b
```

They are used to run a parser and then discard its result. In this setting, these combinators would have the following signatures:

```
(<*) :: Format f xs -> Format f ys -> Format f xs
(*>) :: Format f xs -> Format f ys -> Format f ys
```

This degree of generality cannot be achieved because, due to the double semantics embedded in `Format`, the discarded values must be printed, in order to correctly invert the parser. However it is possible to discard trivial formats, because they do not require any input value. For instance the operator `(<*)` is define as:

```
(<*) :: Format f xs -> Format f '[] -> Format f xs
p <* q =
  case rightIdentityAppend (toSList p) of
    Refl -> p <*> q
```

First of all note that this operator simply combines the two formats with `<*>`, therefore it affects the type level only. However the expression `p <*> q` alone is rejected by the type checker: counterintuitively `xs` does not unify with `xs :++: []`. In order to convince the type checker of this fact, a *proof* about `:++:` is required. The function `rightIdentityAppend` produces for any concrete type level list `xs` the proof that the empty list `'[]` is right identity to append. The proof is by induction on `xs` and hence requires its singleton type:

```
rightIdentityAppend :: SList xs -> xs :++: '[] :~: xs
rightIdentityAppend SNil = Refl
rightIdentityAppend (SCons s) =
  case rightIdentityAppend s of
    Refl -> Refl
```

The data type `a :~: b` represents propositional equality. Pattern matching on it brings into scope the constraint `a ~ b`: a proof for the type checker that `a` is equal to `b`.

```
data a :~: b where
  Refl :: a :~: a
```

A similar reasoning applies for `*>`.

2.2.6 Kleene Operators

In parsing libraries the combinator many corresponds to the Kleene star operator and the combinator some to Kleene plus. Similarly the parser many p succeeds if p succeeds zero or more times and the parser some p succeeds if p succeeds at least once. The default implementation of these combinators is part of the Alternative class.

```
some :: Alternative f => f a -> f [ a ]
some p = (:) <$> p <*> many p

many :: Alternative f => f a -> f [ a ]
many p = some p <|> pure []
```

Their format counterpart has the following signature:

```
many, some :: Format f xs -> Format f (Map [] xs)
```

Their definition follows the same mutually recursive pattern:

```
some f = allCons (toSList f) <$> f <*> many f
many f = some f <|> allEmpty (toSList f) <*> unit
```

The partial isomorphisms allEmpty and allCons have the following signatures:

```
allEmpty :: SList xs -> Iso [] (Map [] xs)
allCons :: SList xs -> Iso (xs :++: (Map [] xs)) (Map [] xs)
```

The isomorphism allCons corresponds to (:) for simple lists. When applied it merges the heads (xs) with the corresponding tails (Map [] xs) applying repeatedly the partial isomorphism cons. When unapplied it splits the given lists (Map [] xs), in heads (xs) and tails (Map [] xs) using the cons deconstructor (unapply). If any of the lists is empty the isomorphism fails. Lastly it appends the two resulting HList.

Similarly the isomorphism allEmpty, when applied, produces an HList of empty lists (Map [] xs), applying repeatedly the partial isomorphism nil. When unapplied produces the empty HList after checking with the deconstructor nil that all the lists are actually empty. If any of them is non-empty the isomorphism fails.

Note that if the parser produced by some and many succeeds, it will produce an heterogeneous list of homogeneous lists, each of the same length. Similarly the corresponding printer will fail if the lists provided have different lengths.

Termination Particular care must be taken when combining many and some with *trivial* formats. Applying either many or some to a trivial format results in a trivial format, because Map [] '[] equals '[]. For example the following *trivial* format denotes a sequence or zero or more spaces:

```
spaces :: Format f '[]
spaces = many (char ' ')
```

The parser for this format shows the desired behavior, however the corresponding printer hangs, without producing any spaces.

In fact the mutually recursive definition of many and some, terminates only under the condition that the repeated action will eventually fail. For parsers this condition can be easily checked. If the parser p consumes at least one token, the parser many p will eventually terminate: since the input string is finite, some p can succeed only a finite number of times, after which only the alternative pure [] will succeed. On the contrary if p does not consume any input the parser many p will hang.

Note that this behaviour is exactly the same for any type instance of Alternative that use the default implementation of many and some:

```
*> many (pure ⊥) :: Maybe [a]
*** Exception: <<loop>>
```

Analogously, when printing the format many p of type Format f (Map [] xs), the lists provided are always finite, therefore the p printer can succeed only a finite number of time and the function will eventually terminate. However when xs is empty, i.e. '[], the *trivial* format p always succeeds, which, combined with many and some leads to non-termination. As a result the spaces printer hangs, while building an infinite list of spaces.

In order to ensure termination also for these formats, the inverse semantics for many and some when combined with *trivial formats* has been adjusted as follows. The function atMost n f k is used to override the behaviour of possibly non-terminating combinators such as many and some, represented by k, precisely allowing to unapply the given format f at most n times.

```
atMost :: Int -> Format f []
       -> (forall xs . Format f xs -> Format f (Map [] xs))
       -> Format f []
atMost n f k = ignore hs <$> (k  (f *> Pure hs))
  where  hs :: HList '[ [a] ]
         hs = hsingleton (replicate n ⊥)
```

Firstly the *trivial* format f is transformed in a *non-trivial* format with f *> Pure hs. As a result also the format k ((f *> Pure hs) is *non-trivial* and thus terminating. Note that the presence of Pure does not affect parsing nor printing, which is instead carried out by f. Using ignore hs the *non-trivial* format is lastly transformed back in a *trivial* format. The list hs contains n bottom objects of type a and it is responsible for the peculiar semantics of atMost. The number of objects contained in this list corresponds to the number of times that the format f is unapplied when printing. Since f is *trivial* it will not inspect these undefined values, therefore, thanks to Haskell lazy semantics, no run-time failure will occur.

The combinators many and some are redefined as follow.

```
many :: Format f xs -> Format f (Map [] xs)
many f =
  case toSList f of
    SNil -> atMost 0 f manyPrim
    _     -> manyPrim f

some :: Format f xs -> Format f (Map [] xs)
some f = case toSList f of
    SNil -> atMost 1 f somePrim
    _     -> somePrim f
```

The functions manyPrim and somePrim correspond to the previous definition of many and some. The semantics of many and some for non-trivial formats is unchanged, while *trivial* formats will be printed f respectively 0 and 1 times. The motivation for this semantics is to output the shortest string that can be matched by the corresponding parser. Furthermore note that the function atMost can also be used to fine-tune arbitrary formats, for example to implement pretty-printers.

Example Consider these two *trivial* formats:

```
spaces0 :: Format f []
spaces0 = many (char ' ')

spaces1 :: Format f []
spaces1 = some (char ' ')
```

When printed they will produce a finite string, respectively containing zero and one space, instead of hanging:

```
*> mkPrinter spaces0 Nil :: Maybe String
Just ""
*> mkPrinter spaces1 Nil :: Maybe String
Just " "
```

2.3 Extensions

In this section several extensions are incrementally added to the core framework discussed previously. They have the purpose to make the library more flexible and usable in practice.

2.3.1 Monadic Format

Formats often need to be self-contained, therefore they include meta-data in the form of tags, magic numbers and headers, needed to decode the rest of a file correctly.

Example The NetPbm is a family of formats that encodes images as a bitmap. The Portable BitMap (PBM) includes a header that contains a magic number (P1) and two numbers n and m that represents the dimensions of the image. The header is followed by a bitmap of of $n \times m$ bits, which encodes the color of each bit (0 for white, 1 for black). In order to parse each row in the bitmap correctly, its dimensions must be taken into account.

A context-sensitive grammar cannot be recognized by Applicative parsers, but requires Monadic parsers, which are strictly more expressive. The need of such class of parsers motivates this extension.

Parsers for context-sensitive grammars are instance of the `Monad` class:

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
  fail :: String -> m a
```

The function `return` lifts a value into the monad and `fail` aborts the computation with an error message. The binding operator `>>=` allows to inspect the value of the first computation and take actions based on it.

The data type `Format` is extended with these new constructors:

```
data Format f xs where
  ...
  Return :: HList xs -> Format f xs
  Fail :: String -> Format f xs
  (>>=) :: Format f xs -> (HList xs -> Format f ys) -> Format f (xs :++: ys)
```

The only remarkable difference is the type signature of the bind constructor. The resulting type-level list is xs :++: ys, instead of only ys, because the input values relative to the first parser must be printed back, in order to correctly invert this parser.

The parser and printing semantics are adapted to include also the Monad constraint and the new cases are defined accordingly.

```
mkParser :: (Monad m, Alternative m) => Format m xs -> m (HList xs)
mkParser (Return hs) = return hs
mkParser (Fail msg) = fail msg
mkParser (f1 >>= k) = do
  hs1 <- f1
  hs2 <- k hs1
  return (happend hs1 hs2)

mkPrinter :: (Monad m, Alternative m) => Format m xs -> HList xs -> m String
mkPrinter (Return _) hs = return ""
mkPrinter (Fail msg) _ = fail msg
mkPrinter (f1 >>= k) hs = (++) <$> mkPrinter f1 hs1 <*> mkPrinter f2 hs2
  where (hs1, hs2) = split (toSList f1) hs
        f2 = k hs1
```

Example The implementation of the PBM format is given in this paragraph as an example of monadic formats. For ease of exposition the text-based encoding is considered.

```
pbmFormat :: Format m '[Int, Int, [[Char]]]
pbmFormat = pbmHeader >>= \(Cons n (Cons m Nil)) -> pBitmap n m
```

The header of a pbm file contains the magic number and two integers that represent the number of rows and columns of the bitmap.

```
pbmHeader :: Format m '[Int, Int]
pbmHeader = p1 *> int <*> (whitespace *> int <* whitespace)
  where p1 = string "P1" *> whitespace
```

Since the magic number is statically know, p1 is a *trivial* format. The format int is a simple formats that converts a non-empty string of digits into an integer. The format whitespace consumes white space characters when parsing and outputs a single space when printing:

```
whitespace :: Format m []
whitespace = some (char ' ' <|> char '\n' <|> char '\r' <|> char '\t')
```

The bitmap is parsed row by row, exploiting the information provided by the header.

```
pBitMap :: Int -> Int -> Format m '[ [[Char]] ]
pBitMap n m = count n (count m (bit <* whitespace))
  where bit = oneOf "01"
```

The format count n f replicates the format f exactly n times.

```
count :: Int -> Format f xs -> Format f (Map [] xs)
count n f
  | n <= 0      = allEmpty (toSList f) <$> unit
  | otherwise = allCons (toSList f) <$> f <*> count (n - 1) f
```

Note that, unlikely many and some, the format combinator count is well-behaved also when applied to *trivial* formats, because it does unapply the format f a finite number of times.

The format oneOf matches any of the characters listed and returns it.

```
oneOf :: [Char] -> Format f Char
oneOf cs = Satisfy (`elem` cs)
```

2.3.2 Token and Stream

Formats are roughly divided in text and binary formats, which determine what type of tokens needs to be recognized and consequently what kind of stream must be provided. As a consequence the framework presented must be adapted to be parametric in the token and stream type.

The Format data type is firstly extended with an additional type parameter that encodes the token type. The constructor Satisfiy is then adjusted accordingly.

```
data Format (f :: * -> *) (i :: *) (xs :: [ * ]) where
  (<$>) :: Iso xs ys -> Format f i xs -> Format f i ys
  Pure :: HList xs -> Format f i xs
  (<*>) :: Format f i xs -> Format f i ys -> Format f i (xs :++: ys)
  Empty :: Format f i xs
  (<|>) :: Format f i xs -> Format f i xs -> Format f i xs
  Satisfy :: (i -> Bool) -> Format f i [ i ]
  Return :: HList xs -> Format f i xs
  Fail :: String -> Format f i xs
  (>>=) :: Format f i xs -> (HList xs -> Format f i ys) -> Format f i (xs :++: ys)
```

Similarly another parameter is added to the ParseSatisfy type class

```
class ParseSatisfy f i where
  satisfy :: (i -> Bool) -> f i
```

In the printing semantics the tokens produced must be combined to produce a stream. The type class PrintToken f i s provides the method printToken, which transforms a single token of type i in a printer of the stream type f s.

```
class PrintToken f i s where
  printToken :: i -> f s
```

Furthermore it is reasonable to expect that the stream type is an instance of Monoid, as it happens for common stream types such as String, ByteString and lists. The type class Monoid provides the methods mconcat and mempty which can be conveniently employed in the semantics of the applicative combinators and monadic combinators.

```
mkPrinter :: (PrintToken f i s, Monoid s, Alternative f, Monad f)
          => Format f i xs -> f s
mkPrinter (Satisfy p) (Cons x Nil)
  | p x       = printToken x
  | otherwise = empty
mkPrinter (Pure _) _   = pure mempty
mkPrinter (f1 <*> f2) hs = mappend <$> mkPrinter f1 hs1 <*> mkPrinter f2 hs2
  where (hs1, hs2) = split (toSList f1) hs
mkPrinter (Return _) _   = return mempty
mkPrinter mkPrinter (f1 >>= k) hs
  = mappend <$> mkPrinter f1 hs1 <*> mkPrinter f2 hs2
  where (hs1, hs2) = split (toSList f1) hs
        f2 = k hs1
```

2.3.3 Extensible Format

The monadic extension discussed in 2.3.1 shows a shortcoming of the current Format representation. In order to provide new primitives the Format data type has been changed, adding new constructors, the semantics functions mkParser and mkPrinter have been adjusted to include the new cases and lastly the set of constraints required to implement them increased. The last change is particularly troublesome, because it brakes the support for non-monadic parsers and printers.

The problem lies in the fact that the data type Format is a *closed universe*, therefore it cannot be arbitrarily extended, without updating existing code. This design problem is known as the expression problem [67] and tests the expressivity of programming language. It requires to define a data type which can be extended adding new cases and adding new functions over it, without recompiling existing code and while retaining static type safety. A number of solutions have been proposed in literature, but an alternative approach to the problem is presented in this section. For simplicity the solution is explained using the original basic example, while the necessary changes to the `Format` data type are shown in section 2.3.4.

Expression Problem The data type `Expr` represents an arithmetic expression:

```
data Expr = Val Int | Add Expr Expr
```

The function `eval` computes the value of an arithmetic expression:

```
eval :: Expr -> Int
eval (Val i) = i
eval (Add e1 e2) = eval e1 + eval e2
```

Other consumer functions can be defined without modifying existing code. For instance the function `pretty` returns a string representation of an expression.

```
pretty :: Expr -> String
pretty (Val i) = show i
pretty (Add e1 e2) = "(" ++ show e1 ++ " + " ++ show e2 ++ ")"
```

However adding a new constructor to `Expr` requires to update all the functions defined over `Expr` to handle the new case.

To solve the expression problem a separate data type is defined for each constructor of the original universe. An additional type parameter is added to each of them.

```
data Val c where
  Val :: Int -> Val c

data Add c where
  Add :: (c a, c b) => a -> b -> Add c
```

The index `c` has kind `* -> Constraint`. The constraint kind is one of the latest extension to the Glasgow Haskell Compiler (GHC), which provides the distinct kind `Constraint`. Equality constraints have kind `Constraint` and type classes are `Constraint` constructors.

Instead of building expressions using the constructors directly, the following smart constructors are used:

```
val :: Int -> Val c
val = Val

add :: (Use a c, Use b c) => a c -> b c -> Add c
add = Add
```

The smart constructor add might look redundant at first sight, however its signature is very important, because it restricts the arguments of Add to be indexed by the same parameter c of kind * -> Constraint. The type synonym Use ensures that the two arguments satisfy the constraints required by Add:

```
type Use a c = c (a c)
```

In general the constraint Use a c requires the presence of an instance for c (a c) for any constraint-indexed type a.

Arbitrary arithmetic expressions can be composed using the smart constructors, while leaving the constraint parameter abstract:

```
foo :: (Use Val c, Use Add c) => Add c
foo = val 0 `add` val 1 `add` val 2
```

The type signature is mandatory: since c is not instantiated, the constraints introduced by the smart constructors must be propagated. As a result each definition will carry a number of class constraints that expose the pieces of expression used. Note however that the user does not have to manually keep track of them, because they are automatically inferred by the type checker, which will trigger a type error if any is missing. Furthermore letting the type-checker infer them is very convenient, for example no duplicated constraints are generated and the order of the constraints themselves is insignificant. For instance in the example foo, the smart constructors val and add are used more than once, however only one constraint per type is inferred.

Semantics functions are defined by means of a properly kinded type class:

```
class Eval a where
  eval :: a -> Int
```

The type class Eval has one parameter of kind *, therefore its kind is * -> Constraint, hence it can be used to instantiate the constraint parameter c.

New instances are defined separately for each data type. For the base cases the parameter c is actually just a phantom type.

```
instance Eval (Val c) where
```

```
  eval (Val i) = i
```

Instead for the recursive cases the parameter `c` must be instantiated with the same type class that is being implemented.

```
instance Eval (Add Eval) where
  eval (Add e1 e2) = eval e1 + eval e2
```

This produces the constraint `c ~ Eval`, which after pattern matching on `Add`, brings into scope the instances `Eval a` and `Eval b`, enabling the recursive calls of `eval` on the two subexpressions. Lastly instead of using `eval` directly an helper function is needed.

```
evalExpr :: Use a Eval => a Eval -> Int
evalExpr = eval
```

This function has the only purpose to fix the constraint parameter `c` of indexed expression to `Eval`. In fact the smart constructors always leave `c` abstract, because the same expression may be interpreted by different semantic functions. Since the parameter of `Eval` is of kind `*`, if `eval` was used directly on a constraint-indexed expressions, its parameter would be ambiguous, resulting in a type error.

Adding new functions Adding new semantic functions is as simple as declaring a new type class and implementing the corresponding instances for each data type.

```
class Pretty a where
  pretty :: a -> String

instance Pretty (Val c) where
  pretty (Val i) = show i

instance Pretty (Add Pretty) where
  pretty (Add e1 e2) = "(" ++ pretty e1 ++ " + " pretty e2 ++ ")"

prettyExpr :: Use a Pretty => a Pretty -> String
prettyExpr = pretty
```

Adding new cases Adding a new case requires to define a new data type and relative smart constructor:

```
data Mul c where
  Mul :: (c a, c b) => a -> b -> Mul c
```

```
mul :: (Use a c, Use b c) => a c -> b c -> Mul c
mul = Mul

bar :: (Use Val c, Use Add c, Use Mul c) => Mul c
bar = foo `mul` foo
```

As required by the expression problem, this solution retains static type safety: when an expression is evaluated by some semantic function, if for some piece of expression the corresponding instance has not been provided, a compile-time error will be triggered.

```
*> evalExpr bar
No instance for (Eval (Mul Eval)) arising from a use of `evalExpr'
  In the expression: evalExpr bar
```

Furthermore no recompilation is needed, because each instance is independent from the others.

Discussion Several solutions to the expression problem have been proposed. A naive solution to the problem consists in defining each case as a separate data type, exploiting parametric polymorphism for the recursive ones.

```
data Val = Val Int
data Add a b = Add a b
```

Likewise semantic functions are defined using type classes.

```
instance Eval Val where
  eval (Val i) = i

instance (Eval a, Eval b) => Eval (Add a b) where
  eval (Add e1 e2) = eval e1 + eval e2
```

The instances for the recursive cases are defined assuming an appropriate context, that brings in scope the instances for the children, therefore allowing the recursive call to eval.

The main drawback of this approach is that the structure of each expression is replicated in its type. For instance:

```
foo :: Add Val (Add Val Val)
foo = Add (Val 1) (Add (Val 2) (Val 3))
```

This encoding is inconvenient not only because of the unwieldy types that it requires, but also because it restricts the class of well-typed programs that can be defined with it. It is easy to incur in infinite types, even in non recursive programs. For instance the following exponentiation function cannot be typed, because it contains an infinite type.

```
exp e n = foldr (\_ -> Mul e) e [1..n-1]
```

Swiestra represents an extensible data type as a fixed-point of a functor [64]. Each constructor is defined as a separate data type that is injected in a functor and semantic functions consist of algebras, defined in a piece-wise manner using the type class system, which are ultimately *folded* over the functor. Swiestra develops automatic injectors, which work as smart constructor and appropriately arrange different data types in a functor. However this feature, essential to make this technique practical, is fragile because it relies on the controversial overlapping instances extension and requires explicit type signatures in order to deduce the right injection.

Carette et al. present an alternative approach to the expression problem that exploits only the type class system [10]. A series of lecture notes [32] contain an introduction closer to this presentation.

Constructors are transformed in methods of a type class, parametrized by a type, that wraps the result of a semantic function.

```
class Expr a where
  val :: Int -> a
  add :: a -> a -> a
```

Semantic functions are defined as a data type that wraps the result of the interpretation and implementing the corresponding instance for `Expr`.

```
newtype Eval = Eval {eval :: Int}

instance Expr Eval where
  val n = Eval n
  add e1 e2 = Eval (eval e1 + eval e2)
```

New cases are added with a new class.

```
class MulExpr a where
  mul :: a -> a -> a

instance MulExpr Eval where
  mul e1 e2 = Eval (eval e1 * eval e2)
```

Compared to the approach presented here, this method has few advantages. Firstly it does not require any particular extension, but exploits uniquely the type class system. Secondly the number of constraints generated can be reduced via subclassing. For instance it would be natural to make the class MulExpr subclass of Expr. One possible disadvantage of this solution is that, once an expression is composed, it cannot be inspected, since methods do not create an actual data type. Naive pattern-match would not work in the solution proposed in this thesis either, however I conjecture that some useful information could be stored with the constraint parameter. Nevertheless both the two solutions fully and effectively solve the original formulation of the expression problem.

2.3.4 Format Revised

The solution proposed in 2.3.3 has been effectively employed to leave the format representation open to extensions.

Functor Format The format data type presented in 2.3.2 have been split in several data types and an additional constraint parameter have been added to each of them. As a representative example of the this transformation, the revised functor format is shown:

```
data FMap c (m :: * -> *) (i :: *) (xs :: [ * ]) where
  FMap :: (c m i a) => Iso xs ys -> a m i xs -> FMap c m i ys
```

The kind of the parameter c is not for the faint of heart:

```
c :: ((* -> *) -> * -> [*] -> *) -> Constraint
```

Luckily kinds are automatically inferred, therefore there is no need to provide an explicit type signature for it[2]. Contrary to the simple example presented in 2.3.3, the parameters m, i and xs of the argument of FMap have been left explicit because the data type itself is indexed over those. Nevertheless the argument is not explicitly indexed by a constraint kind, so that its kind is slightly simpler:

```
a :: (* -> *) -> * -> [ * ] -> *
```

Following the pattern described in 2.3.3, a convenient type synonym is defined:

```
type Use a c m i = c m i (a c)
```

In order to retain the well-known syntax for functors, a smart constructor is also provided:

```
(<$>) :: Use a c m i => Iso args xs -> a c m i args -> FMap c m i xs
f <$> x = FMap f x
```

[2]Unfortunately GHC 7.8 does not support kind synonyms.

Analogous data types and relative smart constructors have been defined for the other core combinators discussed in the previous sections, which include Applicative, Alternative and Monad combinators.

Semantic Functions The interpretation function mkParser has been converted into a type class constraint:

```
class ParseWith (m :: * -> *) (i :: *) a where
  mkParser' :: a m i xs -> m (HList xs)
```

Also in this case a is not explicitly indexed with a constraint parameter, hence an appropriate entry point is needed for constraint-indexed types:

```
mkParser :: Use a ParseWith m i => a ParseWith m i xs -> m (HList xs)
mkParser = mkParser'
```

Furthermore a completely general, yet correct, instance is given in terms of the underlying Functor instance:

```
instance Functor m => ParseWith m i (FMap ParseWith) where
  mkParser' (FMap i f) = apply i <$> mkParser' f
```

The transformation of the semantic function mkPrinter follows exactly the same pattern. Furthermore for the core combinators that belong to the Applicative, Alternative and Monad classes analogous general instances have been provided. As a result the only instances that must be manually added are those of ParseSatisfy and PrintToken, because they are library specific. To give a concrete example, the instance relative to Parsec [35] parser is given:

```
instance Stream s m Char => ParseSatisfy (ParsecT s u m) Char where
  parseSatisfy = satisfy
```

Error Messages Extending the format universe with new cases is straightforward. For example many parsing libraries include a combinator that allows to provide an helpful error message, when a parser fails, typically which token was expected.

```
<?> :: Parser a -> String -> Parser a
```

The combinator is converted in a new data type and an appropriate smart constructor is defined:

```
data Help c m i xs where
  Help :: c m i a => a m i xs -> String -> Help c m i xs
```

```
(<?>) :: Use a c m i => a c m i xs -> String -> Help c m i xs
f <?> msg = Help f msg
```

To keep the library as easily pluggable as possible an appropriate hook is defined, by means of another type class, following the example of the ParseSatisfy and PrintToken classes:

```
class ParseHelp m where
  parseHelp :: m a -> String -> m a
  parseHelp = const
```

Furthermore in this case it is possible to provide an appropriate default behaviour, which simply ignores the given error message. For libraries that provides such combinators, like Parsec [35], the instance is straightforward:

```
instance ParseHelp (ParsecT s u m) where
  parseHelp = (<?>)
```

Once more a generic instance of ParseWith is given, assuming ParseHelp in the context:

```
instance ParseHelp m => ParseWith m i (Help ParseWith) where
  mkParser' (Help f msg) = parseHelp (mkParser' f) msg
```

2.4 Conclusion

2.4.1 Discussion

The library presented in this chapter can effectively describe various real-world formats and derive automatically consistent parsers and printers for them. Text-based and binary formats are both supported by abstracting over the token and stream type. The library relies on a minimal core of basic format combinators, inspired by those employed in parser combinator libraries, that can be easily inverted. As a result the user of this library specifies a data format just like writing a parser for it and gets the inverse printer for free.

Plug-in Framework The library Boomerang [7], currently available on Hackage, provides similar support for invertible parsing and printing. However, differently from the library proposed in this thesis, the parser and printer backend are fixed and implemented from scratch. Considering the abundance of sophisticated, efficient and mature parsing and printing libraries readily available it seems pointless to reinvent yet one more. One of the benefits of this framework is that it allows to reuse any existing library off the shelf, with minimal effort needed from the final user. Arbitrary libraries can seamlessly be plugged in as long as they implement at least the Alternative type class, which is customary.

Extensible Framework The framework is extensible, letting the user tailor the library to his needs, by introducing new primitives. For example some parsing libraries provide specific functions to increase the performance of certain operations, such as skipMany, or to selectively enable backtracking, as it happens with try in Parsec [35]. The format representation discussed in 2.3.4 can be easily extended with new constructs, following the technique explained in 2.3.3.

Alternative semantics The generic parsing and printing semantics given for the applicative combinators are inverse under few assumptions, namely that pure does not consume any input when parsing and that <*> applies the second parser on the input left after the first parser is applied, which is the de-facto standard behaviour in parsing libraries.

Two opposite semantics are common for the choice combinator <|>: greedy and symmetric choice. Each library typically provides only one of them, depending on the specific parsing strategy implemented. For instance Parsec, an industrial strenght parsing library [35], provides a greedy choice operator, whose semantics is predictive parsing with a look ahead of one token. On the other hand UU-Parsinglib, based on the work by Swiestra et al. [63], provides a symmetric choice operator that does not commit to any alternative. If more than one succeeds, then an ambiguous grammar is detected and a run-time failure occurs.

In the Format data type, the combinator <|> is not explicitly marked as greedy or symmetric choice, therefore it is up to the user to ensure that the corresponding printing operator has the appropriate inverse semantics.

2.4.2 Related and Future Work

The problem of unifying parsing and printing in a single specification has been studied extensively in literature. Some work are based on arrows [1, 28]. Alimarine et al. approach consists of writing invertible programs by construction, combining bidirectional arrows, i.e. BiArrows. As an example of this technique they implement a parser using reversible arrows, thus getting the corresponding printer for free. However this technique stands back from the fairly ordinary applicative or monadic style, so common in parsing libraries. In this library formats are described using these established styles, hence it is more likely to be adopted. Boespflug proposes a similar embedded DSL based on reversible combinators called cassettes and that relies on continuation passing style and rank-2 types for composition [41]. The translation to direct style reveals that the continuations are impure and make use of control effects. It is particularly cumbersome and unnatural to define cassette for user-defined data types: code involving continuation passing style is notoriously hard to understand, maintain and debug. Matsuda and Wang develop FliPpr, a quite involved programming transformation system that inverts the specification of a pretty printer and produces the corresponding parser [42]. Similarly to the library developed in this thesis, the system does not reimplement parser and printer from scratch, but rather reuses existing libraries as backend. In addition their technique provides a fine-grained control over the pretty-printing, which however has the drawback of somewhat cluttering the input grammar with pretty printing annotations. For example the *biased choice* operator `<+` separates pretty from ugly patterns, which are nevertheless accepted when parsing. Lastly it is arguably more natural and desirable to implement a parser and get the corresponding printer for free, rather than the other way around. The library proposed in this thesis does not aim to print pretty output directly, however it is still possible to choose one of the existing pretty printing library [27, 68, 62] as backend, for this purpose. Printers for certain formats are inherently ambiguous and result in non-termination, such as the combination of the primitive version of `many` and `some` with *trivial* formats 2.2.6. In these circumstances the function `atMost` tunes a printer and selects a *pretty* version.

There is also a relevant series of works on type-safe variants of C `printf` and `scanf` formatting functions. Danvy's approach consists of an embedded DSL, that exploits continuation passing style [18]. Asai reflects on his work and reveals that it is based on delimited continuations [3]. He then elaborates three new solutions, including two in direct style. Hinze employs functor type-indexed formats, which are combined to compute the types of the expected arguments. The library developed in this thesis is more expressive than common string formats, because it supports user-defined data types and recursive formats. The format representation proposed here can be effectively employed to implement a type safe version of `printf` and `scanf`. However the arguments to the printer are actually collected in a single heterogeneous list: it would be interesting to investigate whether it is possible to provide the `printf` variadic interface, exploiting Haskell advanced type features. While the type of the function can be easily computed from the index of the format using a closed type family, it seems highly non-trivial to curry a function that takes a list-index data type as argument. I conjecture that a continuation-passing style solution could solve the problem.

The library presented in this chapter is greatly inspired by the work on Invertible Syntax Descriptions by Rendel and Ostermann [56]. Following the example of parsing libraries based on applicative functors, they define a small core of invertible primitives, which can be combined to build complex invertible syntax descriptions. Their combinators are defined as methods of ad-hoc classes such as `IsoFunctor` and `IsoApplicative` that resemble the standard Applicative and Functor classes, but are tailored for partial isomorphism. Instead this library, in its simplest description 2.2, represents basic formats as a universe. The parsing and printing semantics are obtained as an *interpretation*, an approach similar to that proposed by Swiestra [49].

Partial Isomorphism The library implemented in this thesis and that of Rendel et al. both rely on partial isomorphisms to couple a function and its inverse in a single entity, however they are slightly different. Their isomorphism is symmetrical, because the two functions are both partial. As a result the corresponding algebra of partial isomorphism is slightly more expressive. For example a partial isomorphism for `foldl` is derived from a minimal set of primitive isomorphisms, as a small-step abstract machine. The partial isomorphism used in this library on the other hand is partial only in the inverse function. The asymmetry precludes the basic combinator `inverse`, essential to derive `foldl`, which is then implemented as a primitive. However, regardless of which partial isomorphism is used, `foldr` could not be derived following the same approach. Since `foldr` is strictly more expressive than `foldl` it is an open question whether the algebra of partial isomorphism is expressive enough to include it. It is also worth pointing out that `foldr` can be inverted only under certain specific conditions.

```
unfoldr g (foldr f z xs) ≡ xs
```

Specifically:

```
g   (f x y)   =   Just (x, y)
g      z      =    Nothing
```

The two isomorphisms differ also in the kind of their parameters: list of types (`[*]`) are used in this library, while Randel employs simple types (`*`). At first sight these representations might look equivalent, because the unit type `()` corresponds to the empty list `'[]`, and nested pairs can be used to collect together heterogeneous types just like type level lists. However it turns out that the type level representation is more precise and therefore more flexible. Consider for example the two alternative signatures of the Applicative sequencing operator:

```
<*> :: f a -> f b -> f (a, b)
<*> :: Format f xs -> Format f ys -> Format f (xs :++: ys)
```

Combining two *trivial* formats `p` and `q` with `<*>` should result in another trivial format, however this happens only with the more accurate list representation:

```
p <*> q :: f ((), ())
p <*> q :: Format f '[]
```

The interaction of trivial formats with list based combinators such as `many` and `some` shows the same wicked behaviour:

```
many p :: f [()]
many p :: Format f '[]
```

The strict kind distinction between target types and container allows to precisely manipulate the former using closed type families such as `Map` and :++:, making the combinators more composable. Furthermore several desired properties of the partial isomorphism algebra, such as associativity, follow directly from the properties of the container type.

Design The technique presented by Rendel et al. is demonstrated on a proof-of-concept parsing and pringing library. The instances given for the newly created type classes, require full access to the library, however it is customary in libraries not to expose fundamental data types, in order to prevent users from breaking internal invariants. The library implemented in this thesis does not rely on a such privileged view, on the contrary, since the behaviour of the core combinators is virtually standard, completely general and reusable instances are given. Furthermore Rendel's library could support generically only monadic parsers, because of the symmetric partiality of the functions stored in his isomorphism. For example consider the `IsoFunctor` combinator `<$> :: Iso a b -> f a -> f b` and the standard `Functor` combinator `<$> :: (a -> b) -> f a -> f b`. In the latter the first argument is a *pure* function, while in their isomorphism both functions are *partial*. It is impossible to apply a partial function and get rid of the `Maybe` wrapper without the increased expressive power of the monadic binding. Concretely the only admissible generic instance that can be given for `IsoFunctor` is:

```
instance Monad f => IsoFunctor f where
  i <$> p = do
    x <- p
    case apply i x of
      Just y -> return y
      Nothing -> fail ""
```

On the other hand, the generic instance given in 2.3.4 for the functor FMap accordingly requires its underlying semantics to be only a Functor. Furthermore using the symmetric partial isomorphism they implement their own satisfy function, named subset :: (a -> Bool) -> Iso a a, and instead assume in the class Syntax a method token that returns the next token in the stream. This design choice does not cope well with existing parsing libraries, which usually provide satisfy as a primitive and implement token as satisfy (const True). An unfortunate consequence of this mismatch is that a generic instance would poorly interact with the underlying library. In this library the isomorphism subset cannot be defined because the apply function is not partial, but more importantly it is not needed, because it is assumed as primitive in ParseSatisfy.

Rendel et al. recognize the problem of providing a suitable interface, in order to support existing libraries and suggest subclassing as the main extension mechanism. This is not an option for our library, because it is not based on type classes, however extensibility is equivalently achieved, by means of a novel solution to the expression problem 2.3.3.

Monadic Formats In section 2.3.1 the format universe is extended with a monadic bind operator. Its signature is not as general as the conventional one for practical purposes. As far as I know this is the first library that provides invertible parsers and printers for context sensitive grammars, even though with some limitations. It would be interesting to investigate further this topic, in order to give to this combinator the standard degree of generality:

```
(>>=) :: Format f xs -> (HList xs -> Format f ys) -> Format f ys
```

The problematic part lies in the printer semantics, in which the dependency embedded by the continuation has to be somehow inverted. Specifically, in order to correctly invert the parser, from an arbitrary HList ys the corresponding HList xs has to be retrieved. The invertible programming paradigm could provide some insight on this problem, though it is important to keep in mind that only injective functions can be inverted, consequently this approach could restrict the class of grammars accepted. On the other hand the solution proposed in the library is conservative, but sound, because no limitation whatsoever is imposed.

Formal Semantics In this thesis the correct invertibility of the formats provided by this library has not been formally proved. It would be interesting to investigate its formal properties, using a proof assistant with dependent types. The work on Total Parser Combinator by Danielsson [16] is particularly relevant and could represent a suitable base for this line of research. Exploiting dependent types and mixing induction and coinduction, he develops a monadic parser combinators library that guarantees termination and supports left recursion. The same techniques are then employed to study correct-by-construction pretty printers [17]. The main result of the paper is the proof that those printers satisfy the round-trip property, i.e. if a value is pretty printed and the resulting string parsed with respect to the same grammar, the original value is obtained. Future work should strive to prove the same result for this format library. On the other hand I expect a weaker property for the converse theorem, in fact the ambiguity problem of the printers discussed in 2.2.6 is resolved arbitrarily. Further research should study a suitable relation $s_1 \subseteq s_2$, which denotes that the representation s_2 is as "pretty" as s_1. The desired theorem would then be `print (parse s)` $\subseteq$ `s`. Analyzing double semantics specification in a proof assistant would also require to precisely describe the inductive and coinductive definitions of the format combinators, hence explaining which formats can be correctly defined in this framework.

Chapter 3

Formal Model

This chapter presents a formal model used to study the semantics of the `diff` and `diff3` algorithms. The model has been developed in the Agda proof assistant [9, 46, 47] and used to mechanically verify several properties of the algorithms. In the presentation minor details such as implicit arguments and `Set` levels will be omitted to improve readability.

3.1 Introduction

3.1.1 Motivation

In practice the semantics of merging algorithms employed in version control systems is not formalized, but it is usually understood empirically, leading to severe misconceptions [30].

For instance it is often hard to predict the outcome of complex merging operations. It is unclear whether a conflict detected is indeed due to two irreconcilable edits or the consequence of a bug. Furthermore, even when merges are successful, they might produce unexpected results, for instance duplicating lines or changing their order. When software artifacts are under revision control, they might produce invalid programs, or, even worse, they could silently alter their semantics [45].

A formal model would identify the specifications of these algorithms, thus unambiguously clearing all these matters. For example the necessary condition discussed in section 3.4.1 sorts out the first issue and the safety properties presented in 3.4.2 address the second.

3.1.2 Characteristics

The basic characteristics of the merger devised in this thesis are listed in this section. They should help the reader to grasp its essential traits, to categorize it and quickly compare it to similar tools.

Structured Data Some tools, such as 3DM by Lindholm, target specific data formats, such as HTML and XML [37, 38], or are specifically designed for software artifacts [2, 70], like those survey by Mens [45]. There are also a number of file systems synchronizers like Unison [4, 54] and that by Ramsey et al. [55]. The algorithms developed in this thesis target structured data more generally, just like those of Chawathe et al. [15, 14]. Specifically it is intended for algebraic data types, represented consequently as typed, ordered rose-trees.

Syntactic Merging *Textual* mergers work directly on files, without taking into account the possible structure of their content. The most widespread tool in this category is GNU diff3. They are by design fairly general, but sometimes imprecise, because of the fixed granularity of the diffs produced [45]. Conversely *syntactic merging* is more precise, because it works on parse trees [45], obtained parsing the input files accordingly to their formats. The merger discussed here performs syntactic merging, applying a variant of the three-way merge algorithm to the nodes of the parse tree. Correspondingly it raises a conflict if the merged object is not well-structured, which in this setting corresponds to an ill-typed term.

Global Alignment Any synchronizer has to find corresponding parts in each replica, i.e. fragments that are somehow related and should be synchronized [22]. This process, called *alignment*, is either *global* or *local*. In the first case the whole replicas are inspected, usually employing a global heuristic to find a good alignment. Conversely in the second case, simple rules are applied locally in specific points of the replicas. For instance Harmony aligns trees whose children have the same names [22]. This works deploys a *global* heuristic, analogous to that of `diff3`, which computes one of the best alignment minimizing an appropriate cost model.

State-based and three-way The merger discussed in this work is *state-based*, i.e. it relies only on the current states of the replicas to be merged, instead of the list of operations that generated them [30]. Furthermore it is *three-way*, so it takes as input two different versions of the same object together with a previous common one, from which they both derived [30]. The main advantage of a state-based approach is that applications are loosely coupled with the synchronizer, which can be used off the shelf [30]. On the other hand, *change-based* techniques, such as that employed by Ramsey et al. [55], require applications to be data-replication aware and to track the operations performed in form of logs. The main benefit of this approach is that the presence of *explicit* operations logs prevents certain kinds of conflict, resulting in a greater number of successful merges.

Persistent Similarly to Harmony [22], the merger proposed in this thesis is *persistent*, i.e. when merging two replicas it will not back out the incompatible changes it may detect, but it will instead report a conflict to the user. Foster et al. remark that persistence precludes *convergence* [22], the guarantee that the merger will always synchronize the two replicas to a same version, at the cost of backing out conflicting edits as needed.

3.1.3 Reasoning by Specification

In Agda it is inconvenient and cumbersome to reason about algorithms directly. Firstly proofs are non-reusable, because they are completely tailored on specific algorithms. Secondly, goals are reduced only following the exact same steps of the algorithm, which leads to overly long and repetitive proofs. This style of reasoning is tiresome, inopportune and obfuscates proofs.

It is preferable, instead, to reason in terms of *specifications*, which can be expressed idiomatically in a data type indexed by inputs and outputs of the algorithm. The advantages of this approach are threefold. Firstly it requires to define the specifications clearly and precisely. Furthermore it fosters reasoning in terms of high-level properties, abstracting from implementation specific details. Secondly it encourages proof reuse, because theorems will be valid for all the algorithms that satisfy the same specifications. Thirdly it allows much easier and intuitive proofs by induction, since it becomes possible to pattern match directly on the specification data type.

This is a general technique that can be employed in similar situations, furthermore it is lightweight because it only requires to show that the algorithm satisfies the specifications set. This approach, sometimes called the graph of the function [8] has been widely employed in this project with positive results, for example in 3.2.3, 3.3.1.

3.1.4 Naming Conventions

The following naming conventions will be consistently used in the rest of the thesis.

- `a b c : Set`
- `as bs cs : List Set`
- `α β γ : F as a` are called *nodes*
- `xs ys zs : DList as`
- `u v w z : Val as bs` are called *values*
- `f g h : u ~> v` are called *edits*, *transformations* or *operations*.

3.2 Basics

This section defines the core concepts of the model. Section 3.2.1 introduces the generic, type-safe representation of data types employed in the model; sections 3.2.2 and 3.2.3 extend the work of Lempsink et al. on type-safe diff and edit scripts [36]. Then section 3.2.4 explains the global alignment strategy deployed and formalizes the merging rules for edits. Lastly sections 3.2.5 and 3.2.6 further elaborate on merging by extending it to edit scripts.

3.2.1 Heterogeneous Rose Trees

Since the ultimate purpose of this work is to detect changes in data types, a generic suitable representation is needed. Algebraic data types are isomorphic to ordered heterogeneous typed trees, in which labeled nodes correspond to constructors and their children to their fields.

The mutually recursive data types `DTree` and `DList` are defined as follows:

```
data DTree : Set -> Set where
  Node : F as a -> DList as -> DTree a

data DList : List Set -> Set where
  [] : DList []
 _::_ : DTree x -> DList xs -> DList (x :: xs)
```

A tree of type `DTree a` represents a value of type `a`; a list of trees of type `DList as` represents a list of `DTree` whose types are determined by `as`. The term `F as a` represents a constructor of an algebraic data type of type `a` that takes arguments of types `as`

The `DTree` encoding is well-typed by construction, because in the signature of `Node`, the same index `as` is shared by `F as a` and `DList as`, therefore representing a well-typed application of a constructor to arguments of the correct type.

Example Consider a data type that represents arithmetic expressions:

```
data Expr : Set where
  One : Expr
  Add : Expr -> Expr -> Expr
```

Its constructors are represented by the following data type[1]:

```
data F : List Set -> Set -> Set where
  One : F [] Expr
  Add : F (Expr :: Expr :: []) Expr
```

The value `Add One One` is encoded as the following tree:

```
two :: DTree Expr
two = Node Add (Node One [] :: Node One [] :: [])
```

For simplicity the data type `F` is kept abstract using a postulate. Furthermore some basic functions to manipulate it are assumed [2].

```
  postulate F : List Set -> Set -> Set
  postulate _=?=_ : (α : F as a) (β : F bs b) -> Dec (α ≡ β)
  postulate eq? : F as a  -> F bs b -> Dec (a ≡ b)
```

Note that it is possible to explicitly implement these features in a type-safe family for closed families of mutually recursive data types, as Lempsink et al. did [36], specifically using modules parametrized by the family of mutually recursive types. I have decided to avoid this encoding to simplify the model.

[1] As shown in this example Agda allows to overload constructors. It should always be clear from the context to which type they refer.

[2] The function `=?=` actually requires heterogeneous equality, because the two nodes have different types.

Utility Functions The signature of two DList utility functions are reported here. Their implementation is straightforward and omitted.

```
_+++_ : DList as -> DList bs -> DList (as ++ bs)
dsplit : ∀ {{as bs}} -> DList (as ++ bs) -> DList as × DList bs
```

The first function appends two DList, while the second function, inverse of the first, splits a list in two parts. In the latter function the lists as and bs are passed as *instance* arguments, a special type of implicit arguments that is automatically resolved at call-sites [19].

3.2.2 Edit Script

An edit script is a list of edit operations that transform the source object into the target object.

Single operations are defined over values, which denote the presence or absence of a node.

```
data Val : List Set -> List Set -> Set where
  ⊥ : Val [] []
  ⟨_⟩ : F as a -> Val as [ a ]
```

The data type is indexed by two lists that respectively contain the types of the fields of a node and its resulting type. Empty values do not store a node, hence their lists are both empty. These two indexes are needed to stack edits in a type safe manner.

The edit operations considered in the model are a superset of the edit operations normally found in GNU diff edit scripts and in that of Lempsink [36]. An edit operation is indexed over two values, which are respectively the *source* and the *target* of the transformation.

```
data _~>_ : Val as bs -> Val cs ds -> Set where
  Nop : ⊥ ~> ⊥
  Del : (α : F as a) -> ⟨ α ⟩ ~> ⊥
  Ins : (α : F as a) -> ⊥ ~> ⟨ α ⟩
  Upd : (α : F as a) (β : F bs a) -> ⟨ α ⟩ ~> ⟨ β ⟩
```

The Nop edit is a no-operation that does nothing at all; the Del α and Ins α edits represent respectively the deletion and the insertion of the node α, and as such the target of the former and the source of latter are ⊥. Lastly Upd α β denotes the update of the node α to β, which concretely represents changing a constructor. Note that when α ≅ β the update is simply a copy.

An edit script collects a finite number of edit operations, while preserving type-safety.

```
data ES : List Set -> List Set -> Set where
  [] : ES [] []
  _::_ : {v : Val as bs} {w : Val cs ds} ->
          v ~> w -> ES (as ++ xs) (cs ++ ys) -> ES (bs ++ xs) (ds ++ ys)
```

Type Safety In the second constructor the prefixes as and cs match the input types of v and w. In the resulting type as and cs are replaced with bs and cs, which are the output types of v and w.

Source and Target object An edit script of type ES xs ys contains the edits that transform a DList xs, called *source*, in a DList ys, called *target*. The *source function* 《 e 》 computes the source of the source of the edit script e:

```
《_》 : ES as bs -> DList as
《 [] 》 = []
《 Nop :: e 》 = 《 e 》
《 Del α :: e 》 with dsplit 《 e 》
... | ds₁ , ds₂ = Node α ds₁ :: ds₂
《 Ins α :: e 》 = 《 e 》
《 Upd α β :: e 》 with dsplit 《 e 》
... | ds₁ , ds₂ = Node α ds₁ :: ds₂
```

Analogously the *target function* ⟦ e ⟧ computes its target object:

```
⟦_⟧ : ES as bs -> DList bs
⟦ [] ⟧ = []
⟦ Nop :: e ⟧ = ⟦ e ⟧
⟦ Del α :: e ⟧ = ⟦ e ⟧
⟦ Upd α β :: e ⟧ with dsplit ⟦ e ⟧
... | ds₁ , ds₂ = Node β ds₁ :: ds₂
⟦ Ins α :: e ⟧ with dsplit ⟦ e ⟧
... | ds₁ , ds₂ = Node α ds₁ :: ds₂
```

Edit scripts manipulate lists of trees rather than single trees, because some operations inherently produce lists of trees [36]. For instance Del α in ⟦_⟧ deletes the node α leaving the list of its children, and similarly for Ins α in 《_》.

3.2.3 Diff

The data type Diff xs ys e is indexed over the *source* list xs and the *target* list ys and the edit script e, and represents the proof that e transforms xs in ys. The two lists are used as stacks, from which arguments for edit operations are popped and results are pushed.

```
data Diff : DList as -> DList bs -> ES as bs -> Set₁ where
 End : Diff [] [] []
 Nop : Diff xs ys e -> Diff xs ys (Nop :: e)
 Del : (α : F as a) -> Diff (xs₁ +++ xs₂) ys e -> Diff (Node α xs₁ :: xs₂) ys (Del α :: e)
 Ins : (α : F as a) -> Diff xs (ys₁ +++ ys₂) e -> Diff xs (Node α ys₁ :: ys₂) (Ins α :: e)
 Upd : (α : F as a) (β : F bs a) -> Diff (xs₁ +++ xs₂) (ys₁ +++ ys₂) e
       -> Diff (Node α xs₁ :: xs₂) (Node β ys₁ :: ys₂) (Upd α β :: e)
```

The base rule End states that the empty edit script transforms the empty source list in the empty target list. Every other rule, one for each edit, appends a different edit to the edit script index and affect the input and target lists accordingly to their semantics: Del consumes the source list, Ins consumes the target list, Upd consumes both and Nop consumes none.

The following result links edit scripts, diff and source and target object.

```
mkDiff : (e : ES as bs) -> Diff 《 e 》 ⟦ e ⟧ e
```

An edit script can be turned into a Diff object in which the source and target objects are given respectively by 《 e 》 and ⟦ e ⟧. The function is defined by induction on the edit script.

Conversely the following theorems show the correspondence between sources and targets of Diff and e.

```
mkDiff《_》 : Diff xs ys e -> xs ≡ 《 e 》
mkDiff⟦_⟧ : Diff xs ys e -> ys ≡ ⟦ e ⟧
```

The proofs are by induction on Diff xs ys e.

It is now evident that Diff xs ys e and Diff 《 e 》 ⟦ e ⟧ e are equivalent representations, hence in other proofs it is possible to freely choose the most convenient. For instance pattern matching directly on terms whose type include the expressions 《 e 》 or ⟦ e ⟧, is usually not possible, because the function application in the indices prevents unification. It is more convenient to introduce the term Diff xs ys e and set xs and ys in the terms, enabling case analysis. Lastly, using these equalities and exploiting rewriting techniques, it is possible to restore the original and more involved statement. See section 3.4.4 for an example.

3.2.4 Merge

The technique devised to merge edits is to apply the three-way merge strategy on values. Informally the three-way merge algorithm compares correspondent sections of two files and of their common ancestor. When the sections of the two files disagree, the version of the ancestor is taken into account. If all of them are different a conflict is detected, otherwise the version that changed from the common ancestor is chosen.

To put this strategy on a formal footing a number of auxiliary definitions are needed. For example it is essential to define precisely the meaning of *corresponding sections*.

Aligned Two edit operations are *aligned* if they share the same source value. Two aligned edits contain at most three distinct values: one common source and two, possibly different, targets, which are treated as corresponding sections. Since the type of an edit uniquely determines its value, enforcing the alignment of two edits is as simple as setting the same source value in their types: no additional data type is required.

Merging is an operation defined over two aligned edits: either it fails raising a conflict, or succeeds producing an edit that comprises both. The following data type represents a successful merge.

```
data _⊔_↧_ : v ~> a -> v ~> b -> v ~> c -> Set where
  Id₁ : (f : v ~> v) (g : v ~> w) -> f ⊔ g ↧ g
  Id₂ : (f : v ~> w) (g : v ~> v) -> f ⊔ g ↧ f
  Idem : (f : v ~> w) -> f ⊔ f ↧ f
```

A value of type `f ⊔ g ↧ h` is the proof that merging `f` with `g` succeeds producing the edit `h`. Each constructor represents a distinct axiom that explains why the merge is possible and determines the merged edit. The rules Id_1 and Id_2 apply when respectively the first and the second transformation is an identity edit. Accordingly to the three-way merge algorithm, when the source node is unchanged in one edit, the other edit is chosen. The fact that merging is an *idempotent* operation motivates the third rule `Idem`. It accounts especially for false-positive conflicts and applies when the same edit is performed independently.

Note that this definition is particularly effective because it is minimal and concise, for instance it does not mention specific edits, yet complete, since it can represent all the true specific merges.

Conflicts are represented by the following data type, indexed by a source value and two target values.

```
data Conflict : (u : Val as bs) (v : Val cs ds) (w : Val es fs) -> Set where
  UpdUpd : (α : F as a) (β : F bs a) (γ : F cs a) -> Conflict ⟨ α ⟩ ⟨ β ⟩ ⟨ γ ⟩
  DelUpd : (α : F as a) (β : F bs a) -> Conflict ⟨ α ⟩ ⊥ ⟨ β ⟩
  UpdDel : (α : F as a) (β : F bs a) -> Conflict ⟨ α ⟩ ⟨ β ⟩ ⊥
  InsIns : (α : F as a) (β : F bs b) -> Conflict ⊥ ⟨ α ⟩ ⟨ β ⟩
```

Conflicts given by InsIns and UpdUpd correspond to conflicting insertions and updates which resemble to some extent the conflicts in the original diff_3. Those given by UpdDel and DelUpd are entirely new and stem from the fact that these two edits are in general non mergeable.

Two incompatible edits give rise to a conflict as described by the next data type:

```
data _⊔_↥_ : (v ~> w) -> (v ~> z) -> Conflict v w z -> Set where
  InsIns : (f : ⊥ ~> ⟨ α ⟩) (g : ⊥ ~> ⟨ β ⟩) (α≠β : ¬ (α ≡ β))
        -> f ⊔ g ↥ InsIns α β
  UpdUpd : (f : ⟨ α ⟩ ~> ⟨ β ⟩) (g : ⟨ α ⟩ ~> ⟨ γ ⟩)  (α≠β : ¬ (α ≡ β))
             (α≠γ : ¬ (α ≡ γ)) (β≠γ : ¬ (β ≡ γ))
        -> f ⊔ g ↥ UpdUpd α β γ
  UpdDel : (f : ⟨ α ⟩ ~> ⟨ β ⟩) (g : ⟨ α ⟩ ~> ⊥) (α≠β : ¬ (α ≡ β))
        -> f ⊔ g ↥ UpdDel α β
  DelUpd : (f : ⟨ α ⟩ ~> ⊥) (g : ⟨ α ⟩ ~> ⟨ β ⟩) (α≠β : ¬ (α ≡ β))
        -> f ⊔ g ↥ DelUpd α β
```

Each constructor includes additionally inequality proofs, essential to make _⊔_↧_ and _⊔_↥_ exclusive. Inequality is logically encoded in Agda as negated equality: the type ¬ P is a synonym for P -> ⊥, where ⊥ is the constructorless data type, which corresponds to falsity under the Curry-Howard isomorphism [69]. Section 3.3.2 include theorems that explicitly rely on them.

The binary operator ⊔ merges two aligned edits. For every pair of edits f and g, it either finds a suitable edit h and provide a proof that f ⊔ g ↧ h, or detects a conflict c, with a proof that f ⊔ g ↥ c.

```
_⊔_ : (f : u ~> v) (g : u ~> w) -> (∃ λ c -> f ⊔ g ↥ c) ⊎ (∃ λ h -> f ⊔ g ↧ h)
```

Table 3.1 schematically outlines its implementation.

3.2.5 Diff_3

The definition of alignment can be naturally extended to edit scripts. Two edit scripts are *aligned* if all their edits are pairwise aligned. The type $e_1 \vee e_2$ denotes total alignment:

f : u ~> v	g : u ~> w	f ⊔ g
Nop	g	Id₁ Nop g
Upd α α	g	Id₁ (Upd α α) g
f	Nop	Id₂ f Nop
f	Upd α α	Id₂ f (Upd α α)
Del α	Del α	Idem (Del α)
Del α	Upd α β	DelUpd (Del α) (Upd α β) α≠β
Upd α β	Del α	UpdDel (Upd α β) (Del α) α≠β
Ins α	Ins α	Idem (Ins α)
Ins α	Ins β	InsIns (Ins α) (Ins β) α≠β
Upd α β	Upd α β	Idem (Upd α β)
Upd α β	Upd α γ	UpdUpd (Upd α β) (Upd α γ) α≠β α≠γ β≠γ

Table 3.1: Implementation of ⊔. f ⊔ g ↥ c, f ⊔ g ↧ h.

```
data _∨_ : ES as bs -> ES as cs -> Set where
  nil : [] ∨ []
  cons : (f : u ~> v) (g : u ~> w) -> e₁ ∨ e₂ -> f :: e₁ ∨ g :: e₂
```

Merging can also be lifted to edit scripts in a similar fashion: it consists in merging each of their aligned edits pointwise.

However, since single merges can fail, the merged edit script may contain conflicts, hence a variant of ES is introduced:

```
data ES₃ : List Set -> Set where
  [] : ES₃ []
  _::_ : {u : Val as bs} -> u ~> v -> ES₃ (as ++ xs) -> ES₃ (bs ++ xs)
  _::ᶜ_ : {u : Val as bs} -> (c : Conflict u v w) -> ES₃ (as ++ xs) -> ES₃ (bs ++ xs)
```

The data type ES₃, contrary to ES, is index over only the input type list, and preserves type-safety only with respect to it. It also contains one additional constructor to include conflicts.

The rules that specify how aligned edit scripts are merged to produce an ES₃, form the following data type:

```
data _⇓_ : e₁ ∨ e₂ -> ES₃ xs -> Set where
  nil : nil ⇓ []
  merge : f ⊔ g ↧ h -> p ⇓ e₃ -> (cons f g p) ⇓ (h :: e₃)
  conflict : f ⊔ g ↥ c -> p ⇓ e₃ -> (cons f g p) ⇓ (c ::ᶜ e₃)
```

The data type is indexed over the global alignment proof object The following type synonym is used instead for greater clarity:

```
Diff₃ : (e₁ : ES xs ys) (e₂ : ES xs zs) {{p : e₁ ∨ e₂}} -> ES₃ xs -> Set
```

```
Diff₃ _ _ {{p}} e₃ = p ⇓ e₃
```

The type Diff₃ e₁ e₂ e₃ is the proof that e₃ is the edit script produced by merging e₁ and e₂. The alignment condition is left implicit using an instance argument.

3.2.6 Merged$_3$

Some of the properties discussed in sections 3.4.2 and 3.4.4 are restricted only to successful Diff₃. This section presents the corresponding specifications.

Definition A Diff₃ is considered *successful* if the merged edit script does not contain any conflict and it is well-typed. In fact a script of type ES₃ as is by-construction well-typed with respect to the source list as, but it may not with respect to the output list.

```
data Merged₃ : ES xs ys -> ES xs zs -> ES xs ws -> Set where
  nil : Merged₃ [] [] []
  cons : f ⊔ g ↧ h -> Merged₃ e₁ e₂ e₃ -> Merged₃ (f :: e₁) (g :: e₂) (h :: e₃)
```

Note that in Merged₃ the third index is of type ES, instead of ES₃ as in Diff₃. It is important to point out that the absence of conflicts does not imply that an edit script is well-typed. For example the following edit script is ill-typed:

```
badExpr :: ES []
badExpr = Ins Add :: Ins One :: []
```

The script produces an ill-typed tree, because Add is applied to only one expression.

The typing judgment e ⇒ as states that the edit script e is well typed and produces a DList as. The typing rules are straightforward:

$$\frac{}{[\,] \Rightarrow [\,]} \qquad \frac{f : v \rightsquigarrow w \qquad w : \mathsf{Val}\ cs\ ds \qquad e \Rightarrow cs \mathbin{+\!+} ys}{f :: e \Rightarrow (ds \mathbin{+\!+} ys)}$$

A trivial inference algorithm can be easily deduced from the typing rules. Moreover note that the conflict cons constructor $(::^c)$ is not mentioned in the typing rules, therefore edit scripts containing conflicts are ill-typed.

Edit scripts of type ES₃ can be converted to ES, if they are well typed:

```
⌜_⌝ : (e : ES₃ xs) -> {{q : e ⇒ ys }}-> ES xs ys
```

The following theorems show that Merged_3 is equivalent to Diff_3 whose merged edit script is well-typed.

```
Merged₃-suf : Diff₃ e₁ e₂ e₃ -> e₃ ⇒ ws -> Merged₃ e₁ e₂ ⌜ e₃ ⌝
Merged₃-nec : Diff₃ e₁ e₂ e₃' -> Merged₃ e₁ e₂ e₃ -> e₃' ⇒ ws -> e₃ ≡ ⌜ e₃' ⌝
```

Discussion It is worth pointing out that, contrary to textual tools, the merger described in this thesis can fail not only with value related conflicts, but also producing an ill-typed term. The choice of indexes for Merged_3 is non trivial and already poses some interesting question:

```
data Merged₃ : ES as bs -> ES as cs -> ES as ? -> Set where
```

What type list should be put instead of the question mark? Clearly when `bs ≡ cs`, the list `bs` is a good choice. However this assumption is too restrictive, because at some point the edit scripts may have different output types, but still there might exist a most general type. A possible alternative would be to choose an asymmetric signature, so that the merged script always shares the output list with the first script.

```
data Merged₃ : ES as bs -> ES as cs -> ES as bs -> Set where
```

Crucially also this choice is not satisfactory, because there could be merges (Id_1), that follow the second script and its types, which would clash with this choice. As a result there is no optimal a priori choice of type that would not be somehow restrictive. Therefore a completely unrelated type `ws` is given as output type and a merged edit script must be typechecked to actually compute it, if one exists.

3.2.7 Summary

This section has introduced the fundamental definitions of the formal model. Section 3.2.1 shows an encoding of algebraic data types as heterogeneous rose trees, i.e. a combination of `DTree` and `DList`, whose nodes correspond to data type constructors. The edit script data type `ES xs ys` is a well-typed list of edits, which transform a source object, a `DList xs` given by `《 e 》`, in a `DList ys` given by `⟦ e ⟧`. The specifications of the `diff` algorithm are given by the `Diff x y e` data type, which is proved to be in a one to one relationship with `Diff 《 e 》 ⟦ e ⟧ e`. Section 3.2.4 defines the notion of *aligned* edit scripts, for which a consistent merge semantics is given. It consists of two data types: `f ⊔ g ↧ h` denotes a successful merge resulting in the edit `h`, while `f ⊔ g ↥ c` denotes two irreconcilable edits that raise a conflict. The binary operator `⊔` merges two aligned edit, producing either of the two proofs. After extending the alignment conditions to whole edit scripts ($e_1 \vee e_2$) the specifications of the diff3 algorithm are given in terms of the $\mathsf{Diff}_3\ e_1\ e_2\ e_3$ data type. The result of diff3 is of type ES_3 xs, a variant of the previous edit script that may contain conflicts and that is type safe only with respect to the input list. Lastly the $\mathsf{Merged}_3\ e_1\ e_2\ e_3$ type refines $\mathsf{Diff}_3\ e_1\ e_2\ e_3$ restricting e_3 to be conflictless and well-typed with respect to the output list.

3.3 Algorithms

This section presents the algorithms diff and diff3 and proves that they satisfy the specifications embodied respectively by Diff and Diff_3.

3.3.1 Diff

A diff algorithm takes as input two objects and outputs an edit script that reports the differences between them. It is convenient to model the edit script as a list of instructions that transform the the first object, named *source*, into the second, named *target*, applying edit operations to their nodes. Furthermore diff finds a *minimal length* edit script, or equivalently the *longest common subsequence* of its inputs [5, 30]. Conventionally edit scripts employ only delete, insert and copy operations, however the model presented here slightly deviates, therefore an appropriate cost function is defined.

```
cost : ES as bs -> ℕ
cost (Nop :: e) = 1 + cost e
cost (Del α :: e) = 1 + cost e
cost (Ins α :: e) = 1 + cost e
cost (Upd α β :: e) = distance α β + cost e
cost [] = 0
```

The edit Nop has weight one, even though it has no operational effect, because it does increase the length of an edit script. In other words for any edit script that contains some Nop an equivalent script with smaller cost can be obtained by removing each Nop edit. The function distance weights the difference between two nodes and it is expected to be a *metric* on the set of nodes.

Metric A metric on a set A is a function $d : (A \times A) \to \mathbb{R}$ such that $\forall x, y, z \in A$:

$$
\begin{aligned}
&d(x,y) \geq 0 && \textit{(non-negativity)}\\
&d(x,y) = 0 \Leftrightarrow x = y && \textit{(coincidence axiom)}\\
&d(x,y) = d(y,x) && \textit{(symmetry)}\\
&d(x,z) \leq d(x,y) + d(y,z) && \textit{(triangle inequality)}
\end{aligned}
$$

A reasonable choice for distance is the *discrete distance*:

$$d(x,y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{otherwise} \end{cases}$$

The binary operator _⊓_ returns the edit script that minimizes cost. Its implementation is straightforward and thus omitted.

```
_⊓_ : ES as bs -> ES as bs -> ES as bs
```

The algorithm proceeds as follows:

```
diff : DList as -> DList bs -> ES as bs
diff [] [] = []
diff [] (Node β ys₁ :: ys₂) = Ins β :: diff [] (ys₁ +++ ys₂)
diff (Node α xs₁ :: xs₂) [] = Del α :: diff (xs₁ +++ xs₂) []
diff (Node α xs₁ :: xs₂) (Node β ys₁ :: ys₂) with eq? α β
... | yes refl = Del α :: diff (xs₁ +++ xs₂) (Node β ys₁ :: ys₂)
               ⊓ Ins β :: diff (Node α xs₁ :: xs₂) (ys₁ +++ ys₂)
               ⊓ Upd α β :: diff (xs₁ +++ xs₂) (ys₁ +++ ys₂)
... | no a≠b   = Del α :: diff (xs₁ +++ xs₂) (Node β ys₁ :: ys₂)
                ⊓ Ins β :: diff (Node α xs₁ :: xs₂) (ys₁ +++ ys₂)
```

When one of the input list is empty the other is consumed by applying repeatedly either the Ins or Del edit to the nodes left. When both are non empty there are three alternatives. Either the input node can be consumed by Del, or the output node can be consumed by Ins, or, if the two nodes have the same type, the first can be mapped into the second by Upd. Among these options, the best script is selected using the ⊓ operator. Note that the Nop edit is never used because it does not consume neither of the two lists: no progress can be made with it.

The algorithm implemented in this way is very inefficient, because it contains several recursive calls that perform the same sub-computations multiple times. The algorithm can be made practical with memoization [36], however the focus here is on correctness, rather than computational complexity, therefore the simpler version will be used.

Technical remark Agda is a total language and therefore requires all the functions to terminate. Since termination is in general undecidable, Agda's termination checker accepts only structural recursion, which safely guarantees termination. In this case diff is rejected as possibly non-terminating, because it is not structurally recursive due to the presence of +++ in the arguments. In the actual model, to overcome this limitation, the function has been adjusted to include an additional parameter, which is an upper-bound on the number of nodes contained in the source and target lists.

```
size : DList as -> ℕ
size [] = 0
size (Node α xs :: ys) = 1 + size xs + size ys

sdiff : {n : ℕ} (xs : DList as) (ys : DList bs) -> size xs + size ys ≤ n -> ES as bs
```

The implementation of sdiff (sized diff) is the same, but recursive calls are structurally recursive on the upper-bound proof object. Minor lemmas that show that size distributes over +++ are needed to prove that the number of nodes yet to be processed is strictly decreasing in the last case.

The function diff then simply calls sdiff using an appropriate upper bound.

```
diff : DList as -> DList bs -> ES as bs
diff xs ys = sdiff {n = size xs + size ys} xs ys (≤-refl (size xs + size ys))
```

The function ≤-refl states that the relation ≤ is *reflexive*, i.e. for any n it follows that n ≤ n. The proof is by induction on n.

Sufficient The lemma used to relate Diff xs ys e and diff xs ys e is the following:

```
Diff-suf : ∀ (xs : DList as) (ys : DList bs) -> Diff xs ys (diff xs ys)
```

Since diff simply calls sdiff the actual proof is by induction on the upper-bound and closely follows its structure. The proof is not particularly interesting and therefore omitted.

The theorem intuitively holds because any edit script e that maps xs into ys satisfies Diff xs ys e. The diff algorithm computes one of them and in particular the one with minimal length. However note that Diff xs ys e does not imply that that e ≡ diff xs ys, because it does not force e to be of minimal length. Choosing a weaker specification was an aware design decision: all the properties considered in the model hold regardless of the fact that the scripts are optimal. The advantage of this choice is that at any time it is possible to change the diff algorithm and the relative properties would still be valid. For example an heuristic that quickly finds a suboptimal edit script may be employed to meet certain performance requirements.

Discussion To minimize the cost function, the diff algorithm strives to match equal nodes, because they have the least distance, similarly to what happens in common diff algorithms [36]. An update involving different nodes is preferred to the equivalent insert and consecutive deletion. This crucial aspect will be explained in section 3.3.2. The no-operation is never preferred, because it consumes neither the source or the target object, but increases the cost of an edit-script. The reason behind introducing this superfluous edit operation will be clarified in 3.3.2.

3.3.2 Diff$_3$

This section introduces the diff$_3$ algorithm used to compare three objects, one of which is considered a previous common version of the other two. Similarly to GNU diff3, the algorithm takes as input three objects and outputs an edit script that combines the changes between them. Likewise the algorithm does not operate on the objects directly, but rather calls the diff subroutine twice to detected the changes from the common version to the others. It then merges the two edit scripts so obtained to produce an edit script that combines both.

The core function is merge$_3$ which combines two *aligned* edit scripts, applying the ⊔ operator to each pair of aligned edits.

```
merge₃ : {e₁ : ES as bs} {e₂ : ES as cs} -> e₁ ∨ e₂ -> ES₃ as
merge₃ nil = []
merge₃ (cons f g p) with f ⊔ g
merge₃ (cons f g p) | inj₁ (c , _) = c ::ᶜ merge₃ p
merge₃ (cons f g p) | inj₂ (h , _) = h ::  merge₃ p
```

To improve readability the following entry point for merge$_3$ is defined:

```
_⊔₃_ (e₁ : ES as bs) (e₂ : ES as cs) {{p : e₁ ∨ e₂}} -> ES₃ as
_⊔₃_  _ _ {{p}} = merge₃ p
```

Sufficient The next result shows that ⊔$_3$ satisfies the specifications set by Diff$_3$.

```
Diff₃-suf : {e₁ : ES as bs} {e₂ : ES as cs} (p : e₁ ∨ e₂) -> Diff₃ e₁ e₂ (e₁ ⊔₃ e₂)
```

The proof follows immediately by induction on $e_1 \vee e_2$.

Complete The following theorem proves that the algorithm is complete with respect to the specification, i.e. that for any triplet of edit scripts e_1 e_2 e_3 that satisfies the Diff_3 specifications, the merged edit script e_3 is the result of $e_1 \sqcup_3 e_2$.

The proof is short, but interesting, therefore it will be reported in full.

```
Diff₃-nec : Diff₃ e₁ e₂ e₃ -> e₃ ≡ e₁ ⊔₃ e₂
Diff₃-nec nil = refl
Diff₃-nec (merge {f = f} {g = g} m q) with f ⊔ g
Diff₃-nec (merge m q) | inj₁ (c , u) = ⊥-elim (mergeConflictExclusive m u)
Diff₃-nec (merge m q) | inj₂ (h' , m') with mergeDeterministic m m'
Diff₃-nec (merge m q) | inj₂ (h' , m') | refl = cong (_::_ h') (Diff₃-nec q)
Diff₃-nec (conflict {f = f} {g = g} u q) with f ⊔ g
Diff₃-nec (conflict u q) | inj₁ (c , u') with conflictDeterministic u u'
Diff₃-nec (conflict u q) | inj₁ (c , u') | refl = cong (_::ᶜ_ c) (Diff₃-nec q)
Diff₃-nec (conflict u q) | inj₂ (h , m) = ⊥-elim (mergeConflictExclusive m u)
```

The proof is by induction on Diff_3 e_1 e_2 e_3. The equivalence is immediate in the base case , when both e_1 and e_2 are empty edit script (nil). In the first recursive case (merge), there are two *aligned* edits f and g that are merged in h, since m has type f ⊔ g ↧ h. The goal is the equivalence h :: e_3 ≡ (f :: e_1) $\sqcup_3$ (g :: e_2). By inductive hypothesis (Diff_3-nec q), it follows that e_3 ≡ e_1 ⊔ e_2, so the only proof obligation left is to show that f ⊔ g ≡ h. By case analysis, either f ⊔ g fails raising a conflict or it succeeds producing a merged edit h'. The first case is discharged by contradiction: merges and conflicts are mutually exclusive. The second case requires to show that the proofs m : f ⊔ g ↧ h and m' : f ⊔ g ↧ h' imply that h ≡ h'. This follows from the property that ⊔ is deterministic [6], i.e. same inputs lead to same outputs. A similar line of reasoning applies in the second recursive case (conflict), in which conflicts determinism is used instead.

The lemmas used in the proof are reported here for completeness.

Mutual exclusion The lemma mergeConflictExclusive asserts that two aligned edits are *exclusively* conflicting or reconcilable.

```
mergeConflictExclusive : f ⊔ g ↧ h -> ¬ (f ⊔ g ↥ c)
mergeConflictExclusive (Id₁ f g) (UpdUpd .f .g α≠β α≠γ β≠γ) = α≠β refl
mergeConflictExclusive (Id₁ f g) (UpdDel .f .g α≠β) = α≠β refl
mergeConflictExclusive (Id₂ f g) (UpdUpd .f .g α≠β α≠γ β≠γ) = α≠γ refl
mergeConflictExclusive (Id₂ f g) (DelUpd .f .g α≠β) = α≠β refl
mergeConflictExclusive (Idem f) (InsIns f f α≠β) = α≠β refl
mergeConflictExclusive (Idem f) (UpdUpd f f α≠β α≠γ β≠γ) = β≠γ refl
```

The proof is by contradiction. Since ¬ (f ⊔ g ↥ c) is short for f ⊔ g ↥ c -> ⊥, one additional parameter of type f ⊔ g ↥ c is included. From f ⊔ g ↧ h and f ⊔ g ↥ c, falsity (⊥) has to be produced.

Case analysis on the two terms leads to incongruent conclusions. Let us examine the first one as an example, the others follow from similar considerations. The term UpdUpd f g α≠β α≠γ β≠γ asserts that f has type ⟨ α ⟩ ~> ⟨ β ⟩ for some nodes α and β. Moreover the term α≠β claims that ¬ (α ≡ β), or equivalently α ≡ β -> ⊥. In the term Id_1 f g, the edit f has type v ~> v for any value v. Since these terms have types f ⊔ g ↧ h and f ⊔ g ↥ c, their edits f are the same, hence their types are unified, assigning the most general type ⟨ α ⟩ ~> ⟨ α ⟩ to f. As a consequence β is actually α, then α≠β has type α ≡ α -> ⊥. By reflexivity (refl), α ≡ α for any α, then applying α≠β to it produces ⊥.

Determinism Even though the ⊔ operator is a binary *function*, in the data types f ⊔ g ↧ h and f ⊔ g ↥ c, the symbol ⊔ is just a part of their identifiers, and as such as nothing to do with the ⊔ operator. These data types merely represents *ternary relations* over their indexes and may or may not be functional. This property has to proved for each of them.

The structure of the proof is standard and requires to show that for any pair of triplets (x, y, z_1) and (x, y, z_2) satisfying the relation, it follows that $z_1 = z_2$.

```
mergeDeterministic : f ⊔ g ↧ h₁ -> f ⊔ g ↧ h₂ -> h₁ ≡ h₂
mergeDeterministic (Id₁ f g) (Id₁ .f .g) = refl
mergeDeterministic (Id₁ f g) (Id₂ .f .g) = edit-≡ g f
mergeDeterministic (Id₁ f .f) (Idem .f) = refl
mergeDeterministic (Id₂ f g) (Id₁ .f .g) = edit-≡ f g
mergeDeterministic (Id₂ f g) (Id₂ .f .g) = refl
mergeDeterministic (Id₂ f .f) (Idem .f) = refl
mergeDeterministic (Idem f) (Id₁ .f .f) = refl
mergeDeterministic (Idem f) (Id₂ .f .f) = refl
mergeDeterministic (Idem f) (Idem .f) = refl
```

The proof follows almost directly from case analysis on the two arguments. It is immediate when either the two constructors is Idem, because it implies that f ≡ g ≡ h, and when the two constructors are the same (Id_1 and Id_1, Id_2 and Id_2). In the spurious cases left (Id_1 and Id_2), the goal is to show that f ≡ g given that they both have type v ~> v. The following lemma shows that two identity edits that shares the same value are indeed equal.

```
edit-≡ : (f g : v ~> v) -> f ≡ g
edit-≡ Nop Nop = refl
edit-≡ (Upd α .α) (Upd .α .α) = refl
```

The proof for f ⊔ g ↥ c follows directly from case analysis on the arguments:

```
conflictDeterministic : f ⊔ g ↥ c₁ -> f ⊔ g ↥ c₂ -> c₁ ≡ c₂
conflictDeterministic (InsIns f g α≠β) (InsIns .f .g α≠β₁) = refl
conflictDeterministic (UpdUpd f g α≠β α≠γ β≠γ) (UpdUpd .f .g α≠β₁ α≠γ₁ β≠γ₁) = refl
conflictDeterministic (UpdDel f g α≠β) (UpdDel .f .g α≠β₁) = refl
conflictDeterministic (DelUpd f g α≠β) (DelUpd .f .g α≠β₁) = refl
```

In order to use the $⊔_3$ operator previously defined, an alignment proof $e_1 \vee e_2$ is needed. If two edit scripts share the same source object, they both will include edits that process its nodes. Furthermore, since edit scripts work in a depth first order, these edits will be found in each of the two edit scripts in the same order. Unfortunately this is not enough to conclude that the two edit scripts are completely aligned, since inserts may occur at any point.

Nevertheless it is possible to realign the two edit scripts, inserting a finite number of Nop edits. It is important to emphasize that, an edit script extended in this way does not affect its semantics with respect to the source and target function. Intuitively this is correct, since the Nop edit has no effect at all in those functions. The following preliminary definitions put this property on a formal footing.

Extension The statement $e_1 \trianglelefteq e_2$ means that e_2 extends e_1 introducing a finite number of Nop edits.

```
data _⊴_ : ES as bs -> ES as bs -> Set where
  stop : [] ⊴ []
  cons : (f : v ~> w) -> e₁ ⊴ e₂ -> f :: e₁ ⊴ f :: e₂
  nop : e₁ ⊴ e₂ -> e₁ ⊴ Nop :: e₂
```

The following lemmas show that the extended script is indistinguishable from the original one, with respect to the source and target function.

```
⊴-《_》 : e₁ ⊴ e₂ -> 《 e₁ 》 ≡ 《 e₂ 》
⊴-⟦_⟧ : e₁ ⊴ e₂ -> ⟦ e₁ ⟧ ≡ ⟦ e₂ ⟧
```

The proofs are by induction on $e_1 \trianglelefteq e_2$ and rely on the fact that Nop affect neither 《_》 nor ⟦_⟧.

The relation $e_1 \sim e_2$ asserts that there are extensions of e_1 and e_2, which are *aligned*.

```
data _~_ (e₁ : ES as bs) (e₂ : ES as cs) : Set where
  Align : e₁ ⊴ e₁' -> e₂ ⊴ e₂' -> e₁' ∨ e₂' -> e₁ ~ e₂
```

It is possible to show that such extensions exist for any e_1 and e_2 originated from the same source.

```
Diff∨ : Diff as bs e₁ -> Diff as cs e₂ -> e₁ ~ e₂
```

The function is defined by induction on the two arguments, but, because of the number of uninteresting cases, it is lengthy and therefore omitted.

Diff$_3$ Now it is possible to provide the conventional diff$_3$ interface, in which the second object is considered the old common version:

```
diff₃ : DList bs -> DList as -> DList cs -> ES₃ as
diff₃ ys xs zs with Diff∨ (Diff-suf xs ys) (Diff-suf xs zs)
diff₃ ys xs zs | Align _ _ p = merge₃ p
```

The diff algorithm is called implicitly by Diff-suf; the edit scripts so obtained are aligned via extension using Diff∨, from which the alignment proof p is extracted and used to finally merge the scripts with merge$_3$. Note that the call to Diff∨ is valid because Diff-suc is invoked with the same xs as first argument.

3.4 Formal Properties

In this section we will study some properties of the algorithms presented. As pointed out in 3.1.3, the properties will be proved using the specifications of the algorithms, rather than the algorithms themselves, so that the proofs will be simpler and furthermore valid for any algorithm satisfying those specifications. The advantages gained with these additional properties are evident: the model becomes stronger, more reliable and predictable.

3.4.1 Conflicts

In order to safely reason about complex merging operations, the necessary and sufficient conditions required to trigger a conflict are pinpointed.

Firstly, some auxiliary definitions will be introduced.

```
data _∈ᶜ_ : Conflict u v w -> ES₃ xs -> Set₁ where
  here : (c : Conflict u v w) -> c ∈ᶜ (c ::ᶜ e)
  there : c ∈ᶜ e -> c ∈ᶜ f :: e
  thereᶜ : (c' : Conflict u' v' w') -> c ∈ᶜ e -> c ∈ᶜ (c' ::ᶜ e)
```

The type c $\in^c$ e denotes that some conflict c is present in the edit script e.

Given Diff_3 e_1 e_2 e_3 this section aims to find the sufficient and necessary conditions on e_1 and e_2, or more concisely on $e_1 \vee e_2$, such that c $\in^c$ e_3. Since conflicts are triggered by incompatible *aligned* edits and more specifically depend on the three values they involve, a data type to refer to them is needed:

```
data Map∨ (u : Val as bs) (v : Val cs ds) (w : Val es fs)
          : e₁ ∨ e₂ -> Set where
  here : (f : u ~> v) (g : u ~> w) -> Map∨ u v w (cons f g p)
  cons : (f : u' ~> v') (g : u' ~> w') -> Map∨ u v w p -> Map∨ u v w (cons f g p)
```

Note that Map∨ u v w p is parametric in the values u, v and w.

A better looking syntax is introduced with a type synonym, mainly to leave the alignment proof implicit, but also to rearrange the order of the parameters and indexes, since in declarations parameters always precede indexes.

```
_,_⊢_~>[_,_] : (e₁ : ES xs ys) (e₂ : ES xs zs) {{p : e₁ ∨ e₂}} ->
                (u : Val as bs) (v : Val cs ds) (w : Val es fs) -> Set
_,_⊢_~>[_,_] e₁ e₂ {{p}} u v w = Map∨ u v w p
```

The statement e_1 , e_2 ⊢ u ~>[v , w] means that, given e_1 and e_2, their aligned source value u is mapped respectively to v in e_1 and to w in e_2.

The minimal conditions for the presence of conflicts are represented by:

```
data Failure (p : e₁ ∨ e₂) : Conflict u v w -> Set where
  InsIns : (α : F as a) (β : F bs b) -> e₁ , e₂ ⊢ ⊥ ~>[ 〈 α 〉 , 〈 β 〉 ] ->
          (α≠β : ¬ (α ≅ β)) -> Failure p (InsIns α β)
  UpdUpd : (α : F as a) (β : F bs a) (γ : F cs a) ->
            e₁ , e₂ ⊢ 〈 α 〉 ~>[ 〈 β 〉 , 〈 γ 〉 ] ->
            (α≠β : ¬(α ≅ β)) (α≠γ : ¬ (α ≅ γ)) (β≠γ : ¬(β ≅ γ)) ->
            Failure p (UpdUpd α β γ)
  UpdDel : (α : F as a) (β : F bs a) -> e₁ , e₂  ⊢ 〈 α 〉 ~>[ 〈 β 〉 , ⊥ ] ->
           (α≠β : ¬(α ≅ β)) -> Failure p (UpdDel α β)
```

```
DelUpd : (α : F as a) (β : F bs a) -> e₁ , e₂  ⊢ ⟨ α ⟩ ~>[ ⊥ ,  ⟨ β ⟩ ] ->
           (α≠β : ¬(α ≡ β)) -> Failure p (DelUpd α β)
```

A value of type Failure p c denotes that in p there are two aligned edits that are incompatible and will trigger the conflict c. Few observations are in order. Firstly Failure is parametric on p of type $e_1 \vee e_2$, which brings into scope e_1, e_2 and the alignment proof p demanded implicitly by e_1 , e_2 ⊢ u ~>[v , w]. Secondly the conflict c is instead an index, since it must be instantiated properly with a different conflict by each constructor. Lastly the inequalities and the conflicting edits specified by each constructor are consistent with those from f ⊔ g ↥ c, just lifted to edit scripts.

Once more a type synonym is used for readability:

```
_,_↥_ : (e₁ : ES xs ys) (e₂ : ES xs zs) {{p : e₁ ∨ e₂}} -> Conflict u v w -> Set
_,_↥_ e₁ e₂ {{p}} c = Failure p c
```

Necessary The following theorem asserts that if e_3 merges e_1 and e_2 and c $\in^c$ e_3, then it follows that e_1 ,e_2 ↥ c.

```
conflict-nec : c ∈ᶜ e₃ -> Diff₃ e₁ e₂ e₃ -> e₁ , e₂ ↥ c
```

The proof is by straightforward induction and thus omitted.

Sufficiency The converse theorem is slightly more involved. The proof is also by induction, but some of the base cases requires further inspection to be discharged by contradiction. The structure is the same for each kind of conflict, therefore only the proof for one of them is listed here.

```
conflict-suf : e₁ , e₂ ↥ c -> Diff₃ e₁ e₂ e₃ -> c ∈ᶜ e₃
conflict-suf (InsIns α .α (here y .y) α≠β) (merge (Idem .y) d) = ⊥-elim (α≠β refl)
conflict-suf (InsIns α β (here x y) α≠β) (conflict (InsIns .x .y α≠β₁) d)
  = here (InsIns α β)
conflict-suf (InsIns α β (cons x y q) α≠β) (merge m d)
  = there _ (conflict-suf (InsIns α β q α≠β) d)
conflict-suf (InsIns α β (cons x y q) α≠β) (conflict u d)
  = thereᶜ _ (conflict-suf (InsIns α β q α≠β) d)
```

The e_1 , e_2 ↥ c data type only reveals the kind of conflict involved, therefore the argument e_1 , e_2 ⊢ u ~>[v , w] is further inspected. The cons constructor is stripped in the recursive calls and the appropriate there constructor is chosen depending on the value of Diff_3, in particular there in conjunction with merge and there^c with conflict. Both conflict and merge show up in the base cases, but only the former is expected. The presence of the latter is bogus: a conflict should occur, yet merge is reported. Two inserts can only be merged if they insert the same node α, but this contradicts α≠β, the proof stored in e_1 , e_2 ↥ c, which asserts that they are in fact different.

3.4.2 Safety

This section introduces some safety requirements for the Diff and $Diff_3$ data type, which act as sanity checks. Firstly some preliminaries definitions are given, which are then used to state and prove these properties.

Membership The type α ∈ ts denotes that the node α is present in the list of trees ts.

```
data _∈_ : ∀ {ys xs a} -> F xs a -> DList ys -> Set where
  here : (α : F as a) -> α ∈ Node α ts₁ :: ts₂
  there : α ∈ ts₁ +++ ts₂ -> α ∈ Node β ts₁ :: ts₂
```

A similar data type denotes membership of an edit in a script.

```
data _∈ₑ_ : v ~> w -> ES xs ys -> Set where
  here : (f : v ~> w) -> f ∈ₑ f :: e
  there : (g : w ~> z) -> f ∈ₑ e -> f ∈ₑ g :: e
```

The following wrapper data type will be often used for the safety properties. The judgment e ⊢ₑ u ~> v means that in e the value u is mapped to v.

```
data _⊢ₑ_~>_ (e : ES xs ys) : Val as bs -> Val cs ds -> Set where
  Nop : Nop ∈ₑ e -> e ⊢ₑ ⊥ ~> ⊥
  Del : (α : F as a) -> Del α ∈ₑ e -> e ⊢ₑ ⟨ α ⟩ ~> ⊥
  Ins : (α : F as a) -> Ins α ∈ₑ e -> e ⊢ₑ ⊥ ~> ⟨ α ⟩
  Upd : (α : F as a) (β : F bs a) -> Upd α β ∈ₑ e -> e ⊢ₑ ⟨ α ⟩ ~> ⟨ β ⟩
```

Core Properties By induction it is possible to relate membership proofs on edit scripts with those about lists of trees.

```
∈-⟦⟧ : {f : v ~> ⟨ α ⟩} -> f ∈ₑ e -> α ∈ ⟦ e ⟧
∈-⟪⟫ : {f : ⟨ α ⟩ ~> v} -> f ∈ₑ e -> α ∈ ⟪ e ⟫
```

It is easy to restrict the edits only to those with, respectively, a target and a source node, giving an appropriate type to the edit f.

These functions are also used to draw the same conclusions with the judgment type:

```
targetOrigin : e ⊢ₑ v ~> ⟨ α ⟩ -> α ∈ ⟦ e ⟧
targetOrigin (Upd α β x) = ∈-⟦⟧ x
targetOrigin (Ins α x) = ∈-⟦⟧ x

sourceOrigin : e ⊢ₑ ⟨ α ⟩ ~> v -> α ∈ ⟪ e ⟫
sourceOrigin (Upd α β x) = ∈-⟪⟫ x
sourceOrigin (Del α x) = ∈-⟪⟫ x
```

Diff Safety These properties are specific to the edit script data type and the source and target functions. Furthermore, since Diff x y e is equivalent to Diff ⟪ e ⟫ ⟦ e ⟧ e, as showed in section 3.2.3, also Diff shares the same properties.

```
noTargetMadeUp : e ⊢ₑ v ~> ⟨ α ⟩ -> Diff x y e -> α ∈ y
noTargetMadeUp p q rewrite mkDiff⟦ q ⟧ = targetOrigin p

noSourceMadeUp : e ⊢ₑ ⟨ α ⟩ ~> v -> Diff x y e -> α ∈ x
noSourceMadeUp p q rewrite mkDiff⟪ q ⟫ = sourceOrigin p
```

These propositions assert that any target and source node in a script comes, respectively, from the target and source object diffed by the edit script itself.

The converse proposition states that a node α that belongs to the target (source) object, is to be found as target (source) value in some edit in the script that converts the former to the latter.

```
noSourceErase : Diff x y e -> α ∈ x -> ∃ (λ v -> e ⊢ₑ ⟨ α ⟩ ~> v)
noTargetErase : Diff x y e -> α ∈ y -> ∃ (λ v -> e ⊢ₑ v ~> ⟨ α ⟩)
```

The proof is on induction on its arguments. Note that these proofs would be much more cumbersome, if stated directly on ⟦ e ⟧ and ⟪ e ⟫.

Diff$_3$ Safety The safety properties just discussed extend naturally to edits in the Diff$_3$ data type. Specifically an edit present in one of the input edit scripts will be found in the merged edit script, given that it does perform a change and that the output script does not contain conflicts. The first requirement is needed because identity edits can be silently ignored in the merge semantics given by f ⊔ g ↧ h, with the constructors Id$_1$ and Id$_2$. On the other hand the second prerequisite is essential, because Diff$_3$ is *persistent*, i.e. it refuses to back out incompatible edits and instead triggers an appropriate conflict. Before stating more formally the theorem, some simple auxiliary data types are introduced.

Firstly an edit does perform a change if it is not the identity edit, or, in other words if the source and the target values are different.

```
data Change (f : v ~> w) : Set where
  IsChange : (v≠w : ¬ (v ≃ w)) -> Change f
```

Secondly the absence of conflicts in a script is guaranteed by the following data type, that does not have any constructor that adds a conflict to the index.

```
data NoCnf : ES₃ as -> Set where
  [] : NoCnf []
  _::_ : (f : v ~> w) -> NoCnf e -> NoCnf (f :: e)
```

The theorem is firstly proved considering an edit present in the first input script ($\mathsf{f} \in_e \mathsf{e}_1$). The result type $\mathsf{f} \in_3 \mathsf{e}_3$ proves the presence of the edit f in the merged script e_3. It corresponds to $\mathsf{f} \in \mathsf{e}$, presented in 3.4.2, just indexed over a script of type ES_3 as instead of one of type ES as bs.

```
noBackOutChanges₁ : Change f -> Diff₃ e₁ e₂ e₃ -> f ∈ₑ e₁ -> NoCnf e₃ -> f ∈₃ e₃
```

The theorem is proved by induction on $\mathsf{f} \in_e \mathsf{e}_1$ and $\mathsf{Diff}_3\ \mathsf{e}_1\ \mathsf{e}_2\ \mathsf{e}_3$.

The same theorem holds for edits that belong to the second edit.

```
noBackOutChanges₂ : Change f -> Diff₃ e₁ e₂ e₃ -> f ∈ₑ e₂ -> NoCnf e₃ -> f ∈₃ e₃
noBackOutChanges₂ c d p q = noBackOutChanges₁ c (⇓-sym d q) p q
```

This is actually a corollary of the previous theorem, since Diff_3 is symmetric in absence of conflicts.

```
⇓-sym : p ⇓ e₃ -> NoCnf e₃ -> (⋎-sym p) ⇓ e₃
⇓-sym nil _ = nil
⇓-sym (merge m p) (h :: q) = merge (⌶-sym m) (⇓-sym p q)
⇓-sym (conflict u p) ()
```

Note that this is not the case, when e_3 does contain some conflicts, because the Conflict data type is asymmetric.

The converse theorem asserts that any given edit in the merged script, comes from one of the input scripts.

```
noEditMadeUp : f ∈₃ e₃ -> Diff₃ e₁ e₂ e₃ -> f ∈ₑ e₁ ⊎ f ∈ₑ e₂
noEditMadeUp (here f) (merge (Id₁ g .f) d) = inj₂ (here f)
noEditMadeUp (here f) (merge (Id₂ .f g) d) = inj₁ (here f)
noEditMadeUp (here f) (merge (Idem .f) d) = inj₁ (here f)
noEditMadeUp (there g p) (merge m d)
  = S.map (there _) (there _) (noEditMadeUp p d)
```

```
noEditMadeUp (there^c c p) (conflict u d)
  = S.map (there _) (there _) (noEditMadeUp p d)
```

The last safety property states that a successful Merged$_3$ produces an edit script, whose output type is a subset of the output types of the input scripts.

The following data type defines precisely such relation:

```
data _⊆_,_ : List Set -> List Set -> List Set -> Set where
  stop : [] ⊆ [] , []
  cons₁ : xs ⊆ ys , zs -> y :: xs ⊆ y :: ys , zs
  cons₂ : xs ⊆ ys , zs -> z :: xs ⊆ ys , z :: zs
  cons₁₂ : xs ⊆ ys , zs -> x :: xs ⊆ x :: ys , x :: zs
  drop₁ : xs ⊆ ys , zs -> xs ⊆ y :: ys , zs
  drop₂ : xs ⊆ ys , zs -> xs ⊆ ys , z :: zs
```

The type xs ⊆ ys , zs denotes that the list xs is a subset of the union of ys and zs. This is obvious in the basic constructor stop, when all the lists are empty. The invariant maintained by the cons constructors is that, for any three lists xs, ys and zs that satisfy this property, whenever an element is added to xs it must be added also to either ys by cons$_1$, or to zs by cons$_2$, or both by cons$_{12}$. Any element can be freely inserted in ys or zs with drop$_1$ and drop$_2$.

The edits stored in a script are applied to the the source and target objects, traversing their nodes in a depth-first order. The function typesOf collects in a list the types of the internal nodes in the same order.

```
typesOf : DList xs -> List Set
typesOf [] = []
typesOf (Node {a = a} α ts₁ :: ts₂) = a :: typesOf ts₁ ++ typesOf ts₂
```

The lemma theorem mixOf asserts that, upon a successful merge, the types of the target object of the merged script is a subset of the types of the target objects of the two input edits.

```
mixOf : Merged₃ e₁ e₂ e₃ -> typesOf ⟦ e₃ ⟧ ⊆ typesOf ⟦ e₁ ⟧ , typesOf ⟦ e₂ ⟧
```

Section 3.2.6 explains that output list of e$_3$ cannot be computed in advance from the indexes of e$_1$ and e$_2$, therefore in the definition of Merged$_3$ it is existentially quantified. This result gives a more precise statement finding a relation between the output types of the three edit scripts. Foster et al. proves a stronger property, namely that the synchronization preserves the schema of the originals revisions [22]. That result cannot be proved in this context because the two indexes are not plain types, but are list of types, that are used in a stack-like fashion and are essential to enforce type-safety.

3.4.3 Maximality

Another important property discussed in this section is maximality [22], which guarantees that all the *changes* from both the scripts are included in the merged script in a Diff₃ run.

The type Maximal e_1 e_2 e_3 asserts that the script e_3 includes all the changing edits from e_1 and e_2, thus they form a maximal triplet.

```
data Maximal : ES xs ys -> ES xs zs -> ES₃ xs -> Set where
 Nil : Maximal [] [] []
 Id₁ : (f : v ~> v) (g : v ~> w) -> Maximal e₁ e₂ e₃
   -> Maximal (f :: e₁) (g :: e₂) (g :: e₃)
 Id₂ : (f : v ~> w) (g : v ~> v) -> Maximal e₁ e₂ e₃
   -> Maximal (f :: e₁) (g :: e₂) (f :: e₃)
 Idem : (f : u ~> v) -> Maximal e₁ e₂ e₃ -> Maximal (f :: e₁) (f :: e₂) (f :: e₃)
```

The constructor Nil is correct because all the three scripts are the same, thus they form a maximal triplet. The constructor Idem reasonably adds the same edit f to each script e_1, e_2 and e_3, that form a maximal triplet, thus preserving it. The presence of the other two constructors is explained because the merged script must retain the *changes* from the input scripts, in order to be maximal. It is then fine to ignore identity edits, since they do not perform any change.

The following theorem shows that a Diff₃ run that triggers no conflict is maximal.

```
Diff₃-maximal : Diff₃ e₁ e₂ e₃ -> NoCnf e₃ -> Maximal e₁ e₂ e₃
Diff₃-maximal nil [] = Nil
Diff₃-maximal (merge (Id₁ f g) p) (.g :: q) = Id₁ f g (Diff₃-maximal p q)
Diff₃-maximal (merge (Id₂ f g) p) (.f :: q) = Id₂ f g (Diff₃-maximal p q)
Diff₃-maximal (merge (Idem f) p) (.f :: q) = Idem f (Diff₃-maximal p q)
```

The proof is straightforward, since the constructors Id_1, Id_2 and Idem of Maximal e_1 e_2 e_3 correspond exactly to those of $f \sqcup g \leadsto h$. In particular note that, after pattern matching on the latter, h gets always instantiated with either f or g.

3.4.4 Structural Invariants

The last property discussed in this section concerns the ordering of the nodes. An edit script transforms lists of trees, traversing and processing their nodes in depth-first order. It represents a mapping from the source to the target, in which nodes of the source are embedded in nodes of the target. The property central to this section is that the embedding preserves the depth-first ordering of the source and target nodes. However nodes are not only mapped (updated) from the source to the target, but they may also be inserted, or removed. Therefore the embedding-preserving property has to be adjusted to take these transformations into account.

This section is structured as follows. Firstly the depth-first ordering on trees is formally defined and a similar relation is defined over edit scripts. Secondly the equivalence between these relations is proved. Lastly the proof that the same property holds for successful merges is presented.

Depth-First ordering A pre-order depth-first traversal consists of firstly visiting the current node, then traversing recursively its subtrees from the leftmost to the rightmost. The traversal induces an ordering on the nodes, so that those that are visited sooner come before those that are processed later. Since the edit scripts manipulate list of trees, instead of single trees, also the definitions about ordering relations will be adjusted accordingly.

The type `ts ⊢ α ⊏ β` denotes that, in the list of trees `ts`, the node `α` comes before the node `β`, according to the depth first traversal.

```
data _⊢_⊏_ : DList xs -> F as a -> F bs b -> Set where
  here : (α : F as a) -> β ∈ (ts₁ +++ ts₂) -> Node α ts₁ :: ts₂ ⊢ α ⊏ β
  there : ts₁ +++ ts₂ ⊢ α ⊏ β -> Node γ ts₁ :: ts₂ ⊢ α ⊏ β
```

The list is used as a stack of trees containing the nodes to be visited. In the base constructor `here`, the node `α` is pushed on the stack and thus comes before any node `β`, that belongs to the stack of nodes yet to be processed. The recursive constructor `there` adds a node `γ` to a list of trees, in which `α` already comes before `β`. Note that the top elements of the stack (`ts₁`) are popped and combined to construct a new tree, rooted respectively in `α` and `γ`, which is then pushed on the stack. In this way it is possible to extend the stack node by node, retaining type-safety.

Edits ordering The next data type defines a similar relation for edit scripts. The type `e` $\vdash_e$ `f ⊏ g` asserts that in the script `e`, the edit `f` precedes the edit `g`.

```
data _⊢ₑ_⊏_ : ES xs ys -> u ~> v -> w ~> z -> Set where
  here : (f : w ~> z) -> g ∈ₑ e -> f :: e ⊢ₑ f ⊏ g
  there : (h : s ~> t) -> e ⊢ₑ f ⊏ g -> h :: e ⊢ₑ f ⊏ g
```

The following lemma shows that the order of two edits is reflected in the source trees in the order of their source nodes.

```
《》-⊏ : (f : 〈 α 〉 ~> v) (g : 〈 β 〉 ~> w) -> e ⊢ₑ f ⊏ g -> 《 e 》 ⊢ α ⊏ β
```

Note that the type of the two edits guarantees that they have a source node. The proof is by induction over `e ⊢ₑ f ⊏ g`.

A symmetric result is proved for ⟦_⟧:

```
⟦⟧-⊏ : (f : v ~> 〈 α 〉) (g : w ~> 〈 β 〉) -> e ⊢ₑ f ⊏ g -> ⟦ e ⟧ ⊢ α ⊏ β
```

The converse lemma cannot be defined so easily. It would have the signature:

```
⊏-《》 : 《 e 》 ⊢ α ⊏ β -> ∃ λ (f : 〈 α 〉 ~> v) (g : 〈 β 〉 ~> w) -> e ⊢ₑ f ⊏ g
```

The problem is that it is not possible to pattern match on `《 e 》 ⊢ α ⊏ β` because of the function application `《 e 》` in its type. The `Diff` data type can remedy, abstracting over that. Furthermore the explicit existential quantification is avoided introducing two auxiliary data types.

```
data _⊢ˢ_⊏_ (e : ES xs ys) (α : F as a) (β : F bs b) : Set where
  source-⊏ : {f : 〈 α 〉 ~> v} {g : 〈 β 〉 ~> w} -> e ⊢ₑ f ⊏ g -> e ⊢ˢ α ⊏ β

data _⊢ᵗ_⊏_ (e : ES xs ys) (α : F as a) (β : F bs b) : Set where
  target-⊏ :  {f : v ~> 〈 α 〉} {g : w ~> 〈 β 〉} -> e ⊢ₑ f ⊏ g -> e ⊢ᵗ α ⊏ β
```

The type `e ⊢ˢ α ⊏ β` asserts that in `e` there are two edits `f` and `g`, whose *source* nodes are respectively `α` and `β`, such that `f` precedes `g` in `e`. Similarly the type `e ⊢ᵗ α ⊏ β` asserts that in `e` there are two edits `f` and `g`, whose *target* nodes are respectively `α` and `β`, such that `f` precedes `g` in `e`.

Note that both the new types are parametric over the edit script and the two nodes. Indeed they simply wrap the `e ⊢ₑ f ⊏ g` and qualify the edits `f` and `g` with appropriate source and target values.

The following two lemmas use the `Diff` data type to convert the first relation to the second.

```
diff-⊏ˢ : Diff x y e -> x ⊢ α ⊏ β -> e ⊢ˢ α ⊏ β
diff-⊏ᵗ : Diff x y e -> y ⊢ α ⊏ β -> e ⊢ᵗ α ⊏ β
```

The proof is by induction on the `⊏` relation and `Diff`, which eventually proves the presence of `α` in `x` and an associated edit in `e` (`Del` or `Upd` in the first lemma, `Ins` or `Upd` in the second). The safety properties discussed in 3.4.2, specifically two variants of `noSourceErase` and `noTargetErase`), are used to show the presence of an appropriate edit involving `β` in the tail of the script.

Order-preserving embedding All the necessary pieces are now available to prove the final theorem, which states that the edit script data type preserves the depth-first order, in the source and target objects.

The theorem is split in two, the first considering only the ordering of the source object, and the second only the target.

The first one states that, given an edit script `e`, if in its source object a node `α` comes before some node `β`, then one of the following holds:

- `α` is deleted in `e`;
- `β` is deleted in `e`;
- There are two nodes `γ` and `φ`, such that the node `α` is mapped to `γ`, the node `β` is mapped to `φ` in `e` and in the target object, the node `γ` comes before `φ`

The type `e ↦ α ⊏ β` denotes that `e` is a (source) order-preserving embedding:

```
data _↦_⊏_ (e : ES xs ys) (α : F as a) (β : F bs b) : Set where
 Del₁ : e ⊢ₑ ⟨ α ⟩ ~> ⊥ -> e ↦ α ⊏ β
 Del₂ : e ⊢ₑ ⟨ β ⟩ ~> ⊥ -> e ↦ α ⊏ β
 Map₂ : e ⊢ₑ ⟨ α ⟩ ~> ⟨ γ ⟩ -> e ⊢ₑ ⟨ β ⟩ ~> ⟨ φ ⟩
       -> ⟦ e ⟧ ⊢ γ ⊏ φ -> e ↦ α ⊏ β
```

The proof exploits a number of results previously discussed, therefore it is reported in full:

```
preserve-↦ : ⟪ e ⟫ ⊢ α ⊏ β -> e ↦ α ⊏ β
preserve-↦ {e = e} p with diff-⊏ˢ (mkDiff e) p
preserve-↦ p | source-⊏ {f = Del α} x = Del₁ (Del α (⊏ₑ-∈₁ x))
preserve-↦ p | source-⊏ {f = Upd _ _} {Del β} x = Del₂ (Del β (⊏ₑ-∈₂ x))
preserve-↦ p | source-⊏ {f = Upd α γ} {Upd β φ} x
  = Map₂ (Upd α γ (⊏ₑ-∈₁ x)) (Upd β φ (⊏ₑ-∈₂ x)) (⟦⟧-⊏ (Upd α γ) (Upd β φ) x)
```

By the lemma `diff-⊏ˢ`, it follows that `e ⊢ˢ α ⊏ β`, i.e. there are two edits `f` and `g` with source nodes respectively `α` and `β`. The definition of `e ⊢ˢ α ⊏ β` restricts the type of `f` and `g` to be `⟨ α ⟩ ~> v` and `⟨ β ⟩ ~> w`, therefore the only possible edits with these types are `Del α`, `Del β`, `Upd α γ` and `Upd β φ` for some nodes `γ` and `φ`, which are correctly reported by pattern matching. When either of them is a delete, one of the `Del` constructors applies, when both are updates the Map₂ is used. The following lemmas are used to convert the relation e ⊢ₑ f ⊏ g in f ∈ₑ e and g ∈ₑ e. The proofs are by straightforward induction and thus omitted.

```
⊏ₑ-∈₁ : e ⊢ₑ f ⊏ g -> f ∈ₑ e
⊏ₑ-∈₂ : e ⊢ₑ f ⊏ g -> g ∈ₑ e
```

The second theorem is symmetrical to the first and thus it will be briefly sketched. It states that, given an edit script e, if in its target object a node α comes before some node β, then one of the following holds:

- α is inserted in e;
- β is inserted in e;
- There are two nodes γ and φ, such that the node γ is mapped to α, the node φ is mapped to β in e and in the source object, the node γ comes before φ

The type e ↤ α ⊏ β denotes that e is a (target) order-preserving embedding:

```
data _↤_⊏_ (e : ES xs ys) (α : F as a) (β : F bs b) : Set where
 Ins₁ : e ⊢ₑ ⊥ ~> ⟨ α ⟩ -> e ↤ α ⊏ β
 Ins₂ : e ⊢ₑ ⊥ ~> ⟨ β ⟩ -> e ↤ α ⊏ β
 Map₂ : e ⊢ₑ ⟨ γ ⟩ ~> ⟨ α ⟩ -> e ⊢ₑ ⟨ φ ⟩ ~> ⟨ β ⟩
       -> 《 e 》 ⊢ γ ⊏ φ -> e ↤ α ⊏ β
```

The proof follows the same structure as preserve-↦ and will not be discussed any further.

```
preserve-↤ : ⟦ e ⟧ ⊢ α ⊏ β -> e ↤ α ⊏ β
```

Corollary The next corollaries assert that Diff is order-preserving. They rely on the fact that Diff x y e is equivalent to Diff ⟦ e ⟧ 《 e 》 e.

```
Diff↦ : Diff x y e -> x ⊢ α ⊏ β -> e ↦ α ⊏ β
Diff↦ d p rewrite mkDiff《 d 》 = preserve-↦ p

Diff↤ : Diff x y e -> y ⊢ α ⊏ β -> e ↤ α ⊏ β
Diff↤ d p rewrite mkDiff⟦ d ⟧ = preserve-↤ p
```

Merged₃ Also successful merges retain the order-preserving property, i.e. the order of the edits in the input scripts is preserved. In the specifications of Diff₃, the input edits are not rearranged, but rather merged pointwise, keeping the original order in the new script, hence intuitively this property ought to hold. Nevertheless identity edits might be dropped from the merged script, according to the semantics of f ⊔ g ↧ h, consequently the theorem states that successful merges are order-preserving with respect to *change edits*. Note also that successful merges do not trigger any conflict, which would also break this property.

```
diff3-⊏₁ : Change f -> Change g -> e₁ ⊢ₑ f ⊏ g -> Merged₃ e₁ e₂ e₃ -> e₃ ⊢ₑ f ⊏ g
diff3-⊏₁ (IsChange v≠w) c₂ (here f o) (cons (Id₁ .f g) q) = ⊥-elim (v≠w refl)
diff3-⊏₁ c₁ c₂ (here f x) (cons (Id₂ .f g) q)
  = here f (noBackOutChangesMerged₁ c₂ q x)
diff3-⊏₁ c₁ c₂ (here f x) (cons (Idem .f) q)
  = here f (noBackOutChangesMerged₁ c₂ q x)
diff3-⊏₁ c₁ c₂ (there a p) (cons m q) = there _ (diff3-⊏₁ c₁ c₂ p q)
```

The proof of this theorem is by induction on $e_1 \vdash_e f \sqsubset g$. The need for the Change f is obvious in the first case, where the merge proof reveals that f is identity and therefore it would be discarded in favour of g in the merged script. This case is ruled out by contradiction, thanks to this additional piece of information. The auxiliary lemma noBackOutChangesMerged₁ is a corollary of noBackOutChanges₁ discussed in 3.4.2.

```
noBackOutChangesMerged₁ : Change f -> Merged₃ e₁ e₂ e₃ -> f ∈ₑ e₁ -> f ∈ₑ e₃
```

Obviously Merged₃ e₁ e₂ e₃ is a subset of Diff₃ e₁ e₂ e₃, because it restricts e₃ to be well-typed and conflictless, therefore the corollary follows directly from this inclusion.

Of course Merged₃ e₁ e₂ e₃ is order-preserving also with respect to the edits of the second script, since it is symmetric.

```
diff3-⊏₂ : Change f -> Change g -> e₂ ⊢ₑ f ⊏ g -> Merged₃ e₁ e₂ e₃ -> e₃ ⊢ₑ f ⊏ g
diff3-⊏₂ c₁ c₂ p d = diff3-⊏₁ c₁ c₂ p (Merged₃-sym d)
```

The lemma Merged₃-sym swaps the two input scripts.

```
Merged₃-sym : Merged₃ e₁ e₂ e₃ -> Merged₃ e₂ e₁ e₃
Merged₃-sym nil = nil
Merged₃-sym (cons m d) = cons (↧-sym m) (Merged₃-sym d)
```

The final corollary asserts that Merged₃ e₁ e₂ e₃ is order-preserving also with respect to the original source object.

```
Merged₃↦ : Diff x y e₁ -> Diff x z e₂ -> Merged₃ e₁ e₂ e₃ -> x ⊢ α ⊏ β -> e₃ ↦ α ⊏ β
Merged₃↦ {e₃ = e₃} d₁ d₂ d₃ p rewrite
  trans mkDiff⟪ d₁ ⟫ Merged₃⟪ d₃ ⟫ = Diff↦ (mkDiff e₃) p
```

The corollary holds because mkDiff⟪ d₁ ⟫ implies that x ≡ ⟪ e₁ ⟫ and from Merged₃⟪ d₃ ⟫ it follows that ⟪ e₁ ⟫ ≡ ⟪ e₃ ⟫.

3.5 Conclusion

The main contributions of this thesis are:

- A data type generic `diff` algorithm for structured data that employs insert, delete, update edits and *global* alignment.
- A state-based, three-way, persistent `diff3` algorithm.
- A formal model to reason about diffing and merging.

In addition a number of results have been mechanically verified using the model, with the Agda proof assistant.

- The necessary and sufficient conditions for the presence of conflicts have been pinpointed.
- Some **safety properties**, which ensure the basic correctness of the algorithms, have been established and proved.
- The `diff3` algorithm is **maximal**.
- The algorithms preserve some well-defined **structural invariants**, such as order-preservation.

3.5.1 Related Work

The great number of publications shows that there is an increased interest in the topics of change detection and merge. There are several tools that target structured data, with a particular focus on the XML format, such as LaDiff [15], MH-Diff [14] and 3DM [66, 37, 38]. Peters gives a comprehensive survey of them and others [50]. Synchronizers represent another closely related field. Relevant examples include Unison, a file synchronizer [4, 54]; Harmony, a synchronizer for heterogeneous data [53, 22] by Pierce et al. and an algebra for file synchronization [55]. Lastly there are many mergers tailored for software artifacts [70, 2]. Mens gives a comprehensive survey of this field[45].

Ordered and Unordered The merger discussed in this thesis is different from all of these, because, contrary to XML and file-systems, the tree-structured objects handled are actually data types, therefore they are *strongly typed* entities. Being typed, the trees are inherently *ordered*, while XML trees could also be unordered. However there is a connection between types and schemas, which Foster et al. consider essential for synchronization [22]. Indeed their synchronizer preserves structural invariants, exploiting schemas to identify specific conflicts, named *schema domain conflicts*. Similarly in this work a merge is considered *successful* only if it is well-typed 3.2.6. Chawathe et al. assume that nodes have unique identifiers [15], which greatly simplifies the change detection phase. However this assumption is too simplistic: data format do not usually mark their content and even XML trees may or may not have identifiers in their nodes. The algorithms discussed in this thesis do not rely on unique identifiers, even though they could be adapted straightforwardly to take advantage of them.

Alignment and Data Structures The alignment strategy used to match parts in two versions plays a central role in the quality of merges.

Khanna et al. explain that GNU diff3 aligns lines computing *stable* and *unstable chunks* from the longest common subsequences between the base and the two new versions, obtained by the `diff` subroutine [30]. Overlapping equal lines are matched and fused in stable chunks, leaving possibly conflicting chunks between them. This technique works well in practice because (in software artifacts) lines are mostly unique and it is restricted to a simple, flat, linear structure. Nevertheless this approach would most likely give poor results for trees. Firstly chunks intuitively align flat data like lists, but they would oddly flatten vertical, structured data like trees. Secondly the implicit assumption that basic values are mostly unique might not hold for certain data formats and therefore it would misalign their content. Therefore the diff discussed in this thesis strives to match single nodes, computing an embedding of the original tree into the other; furthermore it respects their structured nature preserving the relative order of the nodes, in the depth first order traversal, as shown in section 3.4.4. This is achieved by generalizing the copy edit to update and retaining the intuitive three-way merge technique for aligned edits. The merger presented in this thesis employs a *global* strategy that produces an optimal alignment, minimizing an appropriate cost model. Conversely Foster et al. choose a *local* strategy, that associates subtrees by name [22]. Chawathe and Molina reduce the change detection problem to a problem of computing a minimum-cost edge cover of a bipartite graph [15]. They remark that for structured data this problem is NP-hard, therefore they develop an heuristic that consists in pruning the graph induced by two trees following a set of pruning rules and then further discard edges to find a *minimal* edge cover. From the edge cover an edit script can be mechanically derived, however they do not discuss changes merge in their work.

Data Structures Data structures are implemented as algebraic data types, that have always the form of structured ordered trees regardless of the fact that may represent unordered collections such as sets and records. Since the alignment strategy is strongly influenced by the actual shape of the trees, these class of objects will be badly matched and will be poorly merged, probably raising unnecessary conflicts. Foster's synchronizer handles sets and records correctly, but just because their algorithm is schema-driven and internally handles schema keys as sets [22]. On the other hand it performs poorly for lists, because it naively aligns lists by *absolute* positions, due to the local alignment strategy used. For example merging o = [Liz; Jo], a = [Jo] and b = [Liz; Joanna] surprisingly produces [Jo; Joanna], instead of [Joanna], which is correctly reported by the merger presented here.

Edit operations The set of edits considered in the change detection phase plays a fundamental role when merging changes. The edits generated by GNU diff include only insert, delete and copy, which in this thesis has been broadened to update as explained previously. Lindholm includes also the edit move in his synchronizer [37], like Chawathe [15, 14], who additionally introduces also the glue edit that fuses two trees. Lindholm however copy consists of copying multiple nodes [37]. Since node sharing is not explicit this operation could easily produce surprising results, because it could match unrelated nodes. In XML this is mitigated since large sections of text are likely unique, but in our context it is in general an invalid assumption. Move is quite a controversial edit. It can be expressed in the model presented in this thesis by delete and insert, just like in Ramsey's [55], however this sequence of edits might easily clash with possible updates in the other version, raising unnecessary conflicts. Crucially tree rearrangements might not be recognized as a natural edit in certain data formats, but they could for others. For example in a table filled with numbers it would look odd to describe changes in few numbers using moves, but it is perfectly reasonable to do it for rows shuffling. Lindholm takes into account the *context*, made of a node's parent, predecessor and successor, in order to improve the precision in the change detection phase with respect to the move edit [37], however he also remarks that there are few exceptions. It is challenging to craft an alignment strategy that includes the move edit, while keeping its semantics concise and clear, especially in presence of multiple weight parameters, such as in the cost model defined by Chawathe [14]. Furthermore mode edits would weaken the guarantees about the structural invariants described in 3.4.4. Ramsey et al. remark that the *move* operation complicate reasoning, but they suggest to expose it to the user and translate it using delete and insert for reasons of performance, retention of metadata and usability [55]. It is however questionable whether this translation preserves the original meaning.

Formal Model The proliferation of commercial and research file synchronizers and mergers have yielded the need for a formal study of their semantics. Unison [54] is an example of a file synchronizer automacally derived from a formal model [4]. Similarly Harmony is a schema-based synchronization framework [53], whose formal properties have been studied extensively [22] and have inspired the analysis conducted in section 3.4. Ramsey and Csirmaz proposes an algebraic approach to file synchronization [55]. Differently from this thesis, Unison [4] and Harmony [22], their model is operation-based, rather than state-based, which enables an automatic conflict resolution phase named *reconcilation*. Nevertheless the sequence of operations performed at each replica is computed in the update detection phase at each replica comparing the current version with the archive, i.e. the last synchronized version. It is worth pointing out that this phase is greatly simplified by the presence of metadata in filesystems, which helps to disambiguate during the alignment phase and facilitates the change detection phase. They propose a large proof system for a simple filesystem algebra, that is proved to be sound and complete for it. The model is based on a relation between sequences of commands that ensures that a sequence safely approximates another and conflictless synchronization is possible if there is a sequence at least as good as both. Even though the algebra is intuitive, the proof system is quite large and far from concise, to the point that it requires automatic techniques to prove it sound. They claim that reasoning with algebraic laws is more convenient that reasoning directly with the mathematical formulation of the filesystem, however it looks like this approach is unlikely to scale, since a simple model made of five operations requires about fifty laws to be equivalently described algebraically. The combinatorical explosion is due to the fact that laws relate the effect of sequencing two operations. Ramsey's algebraic approach is interesting, but it relies too much in a clear detection of the operations performed, which for structured data is much more blurry. The 3DM tool devised by Lindholm does not include a mechanically verified model and moreover the general merge rules have been derived analyzing the expected result of use cases [37]. His change model is based on *content* and *structural change*. The changes in each version are combined into a change set and can be merged only if the set is *consistent*, i.e. unambiguously determine at most one parent, predecessor and successor for each node. Conflicts are divided in core conflicts and optional conflicts and are overall consistent with those listed in 3.2.4, except for the delete/edit conflict, which is optional, whereas in this thesis is not. It is not explained under which circumstances these changes could be considered compatible. The lack of a formal model raises some doubts about the properties claimed for the merge. First of all the merge is considered symmetric, but later it is admitted that appends of nodes originated from different trees may be accepted in either order. This kind of implementation details are actually relevant from the user point of view and have so far fostered the need for formal, unambiguous models. In this thesis inserts compete for matching positions and may trigger conflicts if incompatible. Secondly the fourth merge rule explicitly requires that changes in either versions to be included in the merged tree, however the fact that updates in deleted trees are optionally considered conflicts contradicts this rule and weaken the property about preservation of edits.

3.5.2 Future Work

This thesis raises several research questions, which ought to be addressed in future works.

Comparison with other tools Even though GNU `diff3` is currently the most widespread merger used for software artifacts, only recently the algorithm has been put on a formal footing by Khanna et al. in [30]. It would be interesting to compare it to the algorithm proposed here and particularly analyze the connection between the conflicts detected by each. It is natural to wonder whether there is any relation between the class of conflicts that they target.

Furthermore since there is an increased interest in these topics, as witnessed by the great number of publications, a general framework to compare properly different tools with similar characteristics should be devised. Specifically it should be possible to determine whether they have the same expressive power and conflict detection capabilities.

Ancestor successor order Section 3.4.4 shows that the depth-first order of the nodes in the source object are preserved in the target object, and also in the merged object, in case of a successful merge. These properties give to the users some indication about the relative position of the nodes in the merged object, however an even stronger property could be enforced. In fact since the data diffed and merged is structured as trees, it would be natural to preserve also the ancestor-successor relationship between nodes. From the user point of view, this relation is certainly clearer and likely more meaningful than the previous one and moreover it fits nicely with the idea of preserving the structure of the objects. Note that this property is stronger and implies the structural invariant discussed in 3.4.4. The edit script data type compares source and target tree, flattening them according to the depth-first order, therefore erasing any vertical relation between their nodes. Consequently the edit script data type is responsible for enforcing the relations between nodes, hence its definition needs to be adjusted to retain the ancestor-successor relationship between nodes.

Hook for data structures As discussed previously tree matching, diffing and merging is inappropriate for algebraic data types that represent unordered data structures. The model presented here could be easily adjusted to allow the definition of specific diffing and merging semantics for certain class of data types.

Move edit and Contexts It would be interesting to optionally extend the model to include move edits, since rearrangements of nodes is common for certain formats such as XML and HTML. As Ramsey remarks, this would likely complicate the formal model and its properties, but it would certainly improve the user experience and the merging capabilities. The idea of node *contexts* described by Lindholm [66] ought to be investigated further especially in presence of a move edit. It particularly fits nicely in this setting, not only because it corresponds to Huet's Zipper data structure [26] and McBride's derivative [43], but also because it allows to exploit the types of the parent, ancestor and successor nodes to align data correctly, producing well-typed trees. Contexts could also be effectively employed in the second part of the alignment phase, in which inserts are aligned with either other inserts, or purposely added no-operations. In fact this naive approach could have a negative impact, in presence of many inserts because too simplistic. In particular contexts could guide this stage, inserting nodes only in valid positions.

Chapter 4

Haskell Implementation

This chapter presents the Haskell implementation of the diff and diff_3 algorithms discussed in 3.

Motivation

Even though Agda's type system is sophisticated, it is still a research prototype and its compiler is still rather immature and does not produce efficient code. Consequently a more mature programming language has been chosen to provide a practical implementation of the algorithms studied in this thesis. Haskell, a general-purpose, strongly typed, purely functional programming language [25, 40] has been used for this purpose. Even though full-fledged dependent types are not available, the Glasgow Haskell Compiler (GHC) [23], the current state-of-the-art, optimizing compiler for Haskell, provides several extensions to the type system, which allow to partially simulate dependently typed programming. The implementation discussed in this chapter relies heavily on them, in particular on Generalized Algebraic Data Types (GADT) [52, 51, 59] and Type Families [13, 12, 58, 60, 33], and was developed using the latest stable release of GHC, version 7.8.3. The shortcomings of the implementation with respect to the formal model, developed instead in a dependently typed language, will be pointed out.

4.1 Basics

This section presents the basic data types and type classes used in the algorithms section.

4.1.1 Type Manipulation

Proxy Programs that inspect and manipulate types need specific data types and functions, whose main role is to fix types and ultimately drive type inference, rather than storing and computing values. For example the data type `Proxy a` is a poly-kinded proxy type that has only one non-bottom value, namely `Proxy`, and it is usually used to specify types in signatures.

```
data Proxy t = Proxy
```

Type Equality In the formal model a decidable type equality operator was assumed in the `diff` algorithm and extensively used to type check merged edit script. Such an operator is provided by the library `Typable` [34]:

```
eqT :: (Typeable a, Typeable b) => Maybe (a :~: b)
```

The class `Typeable` includes the method `typeRep`, that given a proxy type produce a unique type representation for it. The function `eqT` compares the type representations and if they are equals manufactures an equality proof, using unsafe operations. The implementation is then safe under the assumptions that the type representations generated are unique. Instances of the class `Typable` can be automatically generated for any data type and, to further increase the safety of the library, manual instances of `Typeable` are rejected. Type representations include hash fingerprints which are generated accessing GHC internal representations of data.

For convenience the following auxiliary function will be used to compare types:

```
tyEq :: (Typeable a, Typeable b) => Proxy a -> Proxy b -> Maybe (a :~: b)
tyEq _ _ = eqT
```

4.1.2 Universe

The type class `Diff a` denotes that the type `a` can be diffed.

```
class Typeable a => Diff a where
  type FamilyOf a :: [ * ] -> * -> *
  (=?=) :: F xs a -> F ys a -> Maybe (xs :~: ys)
  distance :: F xs a -> F ys a -> Double
  argsTy :: F xs a -> TList xs
  toDTree :: a -> DTree a
  fromDTree :: DTree a -> a
  string :: F xs a -> String
```

The associated type FamilyOf a is to be instantiated with the concrete data type that represents the constructors of a. It corresponds to the postulated data type F of kind List Set -> Set -> Set assumed in the formal model in 3.2.1. For ease of explanation and to keep the implementation consistent with the formal model, the following type synonym is used:

```
type F xs a = (FamilyOf a) xs a
```

A value of type F as a represents a concrete constructor of a that takes arguments of types determined by as. The first method =?= tests for equality between nodes, analogously to the operator described in the formal model, except with a less refined signature. The second method distance assigns a numeric distance between constructors of the same type. It is expected to satisfy the metric axioms listed in 3.3.1. The method argsTy reifies the types of the arguments of a constructor and are needed to type check merged edit scripts. The methods toDTree and fromDTree are used to convert raw data types to a well-typed generic tree representation and the other way around. Lastly the method string returns a string representation of a constructor and it is used exclusively to interact with the user.

In addition the auxiliary function decEq tests whether two constructors belong to the same type.

```
decEq :: (Diff a, Diff b) => F xs a -> F ys b -> Maybe (a :~: b)
decEq _ _ = tyEq Proxy Proxy
```

4.1.3 Typed List

The algorithms manipulate several different kind of typed list, which are all small variation of the heterogeneous list HList introduced in 2.1.2. This section defines them and explains their role.

TList The type TList as represents a list of types as each of them belonging to the family Typeable.

```
data TList as where
  TNil :: TList []
  TCons :: Typeable a => Proxy a -> TList as -> TList (a : as)
```

The presence of the proxy is to conveniently manipulate the type later on.

An appropriate TList can be automatically built for lists of types known at compile time, in a similar manner to what happens for SList with KnownSList, as described in 2.1.2.

```
class KnownTList as where
  tlist :: TList as
```

The only two generic instances are:

```
instance KnownTList '[] where
  tlist = TNil

instance (Typeable a, KnownTList as) => KnownTList (a : as) where
  tlist = TCons Proxy tlist
```

This technique exploits the automatic instance resolution that happens while type-checking, to progressively build the desired TList. In the second instance the type checker infer the expected type Proxy a for Proxy and the recursive call to tlist is justified by the constraint KnownTList as in the context. The constraint Typeable a is instead required by the constructor TCons.

It is also entirely straightforward to provide an instance for the type class Reify, which converts a TList as into the corresponding singleton type SList as, introduced in 2.1.2.

```
instance Reify TList where
  toSList TNil = SNil
  toSList (TCons _ t) = SCons (toSList t)
```

DList The type DList as represents a list of DTree of types determined by as. Their definition is entirely similar to the one given in the formal model, except that it requires an instance of Diff for each type contained.

```
data DList xs where
  DNil :: DList []
  DCons :: Diff a => DTree a -> DList as -> DList (a : as)

data DTree a where
  Node :: F as a -> DList as -> DTree a
```

The data type DTree a is a type safe representation of an algebraic data type of type a, where the term of type F as a is a reifyed witness of a constructor of a.

4.2 Diff

This section describes the implementation of the diff algorithm presented in section 3.2.3 and the related data types.

4.2.1 Edit Script

The edit script data type described in 3.2.2 consists in a well-typed list of edits of type u ~> v, each of them encoding the change made with their source and target values. That definition is particularly elegant because it isolates the actual set of edits from the conditions required to stack them in a type-safe manner. They are embedded in the cons constructor signature and consist in expecting the input types of the edit to match the prefix of the type lists of the rest of the edit script. Remarkably these rules are the same regardless of the concrete edit at hand, which makes it possible to separate the two data types. This modular representation is unfortunately not possible in the current version of GHC for two reasons. Firstly GADTs are unpromotable, i.e. it is not possible index a GADT by another GADT, which prevents the edit data type to be indexed by values. As Yorgey et. al explain in [71], this feature would require kind equality and kind coercions, which would dramatically complicate type equivalence. Secondly not even indexing edits directly by their input and output type lists would completely solve the issue:

```
data Edit as bs cs ds where
  Ins :: F as a -> Edit [] [] as [ a ]
  Del :: F as a -> Edit as [ a ] [] []
  Upd :: F as a -> F bs a -> Edit as [ a ] bs [ a ]

data Edits xs ys where
  ENil :: Edits [] []
  ECons :: Edit as bs cs ds -> Edits (as :++: xs) (cs :++: ys)
         -> Edits (bs :++: xs) (ds :++: ys)
```

The definition of Edits is rejected because the type parameters xs and ys are ambiguous. The problem is that in general it is not possible to invert type families, therefore the type checker refuses to find suitable parameters xs and ys, to solve unification problems such as zs = as :++: xs, for some given zs. This issue could be resolved including two singleton lists SList xs and SList ys in ECons, which would however slightly obfuscate the code. Note that this is not an issue in Agda data types because parameters must all be declared either as implicit or explicit arguments and hence are always available, which is the essence of the fix proposed using singleton lists.

To sidestep all these limitations the edit script data type have been defined similarly to that of Lempsink et al. in [36], collapsing edits and edit script into a single data type and introducing specific constructors for each kind of edit. Note that this definition completely preserves type safety, but it is only more repetitive.

```
data ES xs ys where
  End :: ES [] []
  Ins :: Diff a => F xs a -> ES ys (xs :++: zs) -> ES ys (a ': zs)
  Del :: Diff a => F xs a -> ES (xs :++: ys) zs -> ES (a ': ys) zs
  Upd :: Diff a => F xs a -> F ys a -> ES (xs :++: zs) (ys :++: ws)
      -> ES (a ': zs) (a ': ws)
```

4.2.2 Memoization

The diff algorithm described in 3.3.1 is inefficient because the same subcomputations are recomputed multiple times. An equivalent, but more efficient, version can be achieved using memoization, i.e. storing the result of subcomputations in a lookup-table, encoded by the following data type:

```
data EST xs ys where
  NN :: ES [] [] -> EST [] []
  NC :: Diff b => F xs b -> ES [] (b : ys)
     -> EST [] (xs :++: ys)
     -> EST [] (b : ys)
  CN :: Diff a => F xs a -> ES (a : ys) []
     -> EST (xs :++: ys) []
     -> EST (a : ys) []
  CC :: (Diff a, Diff b) => F xs a -> F ys b
     -> ES (a : zs) (b ': ws)
     -> EST (a : zs) (ys :++: ws)
     -> EST (xs :++: zs) (b : ws)
     -> EST (xs :++: zs) (ys :++: ws)
     -> EST (a : zs) (b : ws)
```

A table of type EST xs ys contains an edit script of minimal cost of type ES xs ys, and the subtables corresponding to its tail, obtained by placing either an insert, a delete, or an update edit. The table is partitioned in four groups depending on the fact that the source and target list is empty or not. Specifically NN contains the only edit script in which both of them are empty, in CN and NC the target and source lists are respectively empty and hence contain only one subtable, lastly in CC the lists are both non-empty. The edit script contained in a table can be easily retrieved:

```
getDiff :: EST xs ys -> ES xs ys
getDiff (NN e) = e
getDiff (NC _ e _) = e
getDiff (CN _ e _) = e
getDiff (CC _ _ e _ _ _) = e
```

The function diffT builds a memoization table recursively:

```
diffT :: DList xs -> DList ys -> EST xs ys
diffT DNil DNil = NN End
diffT (DCons (Node a as) xs) DNil = CN a (Del a (getDiff d)) d
  where d = diffT (dappend as xs) DNil
diffT DNil (DCons (Node b bs) ys) = NC b (Ins b (getDiff i)) i
  where i = diffT DNil (dappend bs ys)
diffT (DCons (Node a as) xs) (DCons (Node b bs) ys) = CC a b (best a b i d u) i d u
  where u = diffT (dappend as xs) (dappend bs ys)
        i = extendI a xs u
        d = extendD b ys u
```

The only interesting case is the last one, in which both the input lists are non-empty. Firstly note there is only *one* recursive call to diffT, which is well-founded because both the input lists are consumed removing the nodes a and b. From the table so obtained, the alternative tables in which a is inserted or b is deleted are obtained using the functions extendI and extendD. Among these three options, the script with minimal cost is chosen by the function best.

```
best :: (Diff a, Diff b)
     => f as a -> f bs b
     -> EST (a : xs) (bs :++: ys)
     -> EST (as :++: xs) (b : ys)
     -> EST (as :++: xs) (bs :++: ys)
     -> ES (a : xs) (b : ys)
best f g i d c =
  case decEq f g of
    Just Refl -> Upd f g (getDiff c) & a & b
    Nothing -> a & b
  where a = Del f (getDiff d)
        b = Ins g (getDiff i)
```

Both nodes can be consumed by an update only if they have the same type, which is tested by the decEq function. The other alternatives, which are always possible, consists of deleting the first node or inserting the second. Note that the source and target lists of the three tables are the same of their edit script, hence the scripts produced are all well-typed.

The binary operator & selects the edit script with minimal cost:

```
(&) :: ES xs ys -> ES xs ys -> ES xs ys
x & y = if cost x <= cost y then x else y
```

The function cost computes the score of an edit script, according to the cost model described in 3.3.1.

```
cost :: ES xs ys -> Double
cost End = 0
cost (Ins x xs) = 1 + cost xs
cost (Del x xs) = 1 + cost xs
cost (Upd f g xs) = distance f g + cost xs
```

Of course the cost function inefficiently recomputes the cost of an edit script multiple times. It is entirely straightforward to adjust the edit script data type or the memoization table to store also the score of each edit script. I have decided to omit this optimization from the presentation in order to keep it clear and focused on the type related issues, which are more interesting and challenging.

The function extendI use an auxiliary intermediate data type DES used to existentially quantify the portion of the table that needs to be extended.

```
data DES a xs ys where
  DES :: F zs a -> ES (a : xs) ys -> EST (zs :++: xs) ys -> DES a xs ys
```

If the source list is non-empty, then it is possible to produce such a data type.

```
extractD :: EST (a : xs) ys -> DES a xs ys
extractD (CN g e i) = DES g e i
extractD (CC f g e _ i _) = DES f e i
```

Note that the type a : xs in the target list restrict the possible values of EST only to those listed.

Finally extendI is defined as follows:

```
extendI :: Diff a => F xs a -> DList ys -> EST zs (xs :++: ys) -> EST zs (a ': ys)
extendI f _ i@(NN e) = NC f (Ins f e) i
extendI f _ i@(NC _ e _) = NC f (Ins f e) i
extendI f _ i@(CN _ _ _) =
  case extractD i of
```

```
    DES g e c -> CC g f (best g f i d c) i d c
      where d = extendI f ⊥ c
extendI f _ i@(CC _ _ e _ _ _) =
  case extractD i of
    DES g e c -> CC g f (best g f i d c) i d c
      where d = extendD f ⊥ c
```

First of all the second argument of type `DList ys` is never inspected and it is introduced only to avoid the ambiguity problem discussed previously. Extending the target list does not affect the source list, hence the tables `NC` and `CC` keep the same constructor. On the other hand when the target list is empty, in the `NN` and `CN` case, the constructor is substituted respectively by `NC` and `CC`. In the last two cases the function `extractD` is used to extract the appropriate node `g`, edit script `e` and table `c`, needed to compute the recursive extension `d` and lastly select the table with minimal cost.

The function `extendD` is analogous and thus omitted.

4.2.3 Algorithm

Using all the constructs defined previously, the conventional interface of GNU diff can be easily made available to the user:

```
gdiff :: (Diff a, Diff b) => a -> b -> ES '[ a ] '[ b ]
gdiff x y = getDiff (diffT dx dy)
  where dx = DCons (toDTree x) DNil
        dy = DCons (toDTree y) DNil
```

The inverse function `patch`, which computes the target object from the edit script, is also part of the interface provided to the user.

```
patch :: ES xs ys -> DList ys
patch (Ins x e) = insert x (target e)
patch (Del x e) = target e
patch (Upd x y e) = insert y (target e)
patch End = DNil
```

The function `insert` pops the arguments of the constructor from the given `DList` and builds a well-typed `DTree` which is then pushed on the list.

```
insert :: Diff a => F xs a -> DList (xs :++: ys) -> DList (a ': ys)
insert x ds = DCons (Node x ds1) ds2
  where (ds1, ds2) = dsplit (reifyArgs x) ds
```

The function `dsplit` is entirely similar to the function `dsplit` described in 3.2.1, the first argument is a singleton type needed due to the strict phase separation enforced in Haskell. The function reifyArgs retrieves the singleton type relative to its arguments.

```
reifyArgs :: Diff a => F xs a -> SList xs
reifyArgs = toSList . argsTy
```

4.2.4 Discussion and Related Work

The edit script data type and the memoization technique employed in the diff algorithm presented in this section are largely inspired by the work on type-safe diff by Lempsink [36]. As remarked in section 3.5.1, the update edit used represents a generalization of the original copy edit, and has been preferred because it improves the alignment of nodes, simplifying the merging phase in the diff3 algorithm. The diff algorithm requires to write some boilerplate code, about the same amount of that by Lempsink. In particular for any type involved an instance of Diff have to be provided, which requires to define a data type that encodes its constructors. The code for most of the methods of Diff is entirely straightforward and can be automatically generated using a preprocessor or a meta-programming library such as Template Haskell [61]. Only the implementation of distance is in general domain specific and it must adhere to certain axioms and thus should be provided manually by the user.

Kind system The enrichment of the *kind* system proposed in [71] and available from GHC 7.6 makes the data type definitions "well-kinded" and reduce the additional machinery needed to deal with them. Specifically using the DataKinds extension lists are automatically promoted and allows to index the ES and DList data types with lists of types, whose kind is [*]. Direct pattern match is possible and does not required additional class constrains, such as IsList. Furthermore the kind system has been enhanced so that the correct kind is automatically inferred, reducing the number of kind annotations needed. Most of the time kinds annotations are superfluous and some have been added exclusively for documentation purposes. The memoization related functions have been greatly simplified using auxiliary intermediate GADTs to existentially quantify variables, as opposed to the continuation style employed by Lempsink, in which functions take an higher rank function as an additional parameter. The two approaches are roughly equally expressive, however direct style code is much more readable and concise. As a result the implementation proposed in this thesis requires less boilerplate code and it is closer to the Agda version, showing how dependently typed programming is becoming more and more natural in Haskell.

Change Detection The technique employed by Lempsink et al. in the change detection phase for structured data is a variation of that proposed by Lozano and Valiente to solve the Maximum Common Embedded Subtree problem [39]. The input trees are flattened to a list of nodes according to the depth-first perorder traversal and then the longest common subsequence of them is computed, similarly to what happens in GNU diff. The algorithm proposed in this section follows a similar technique, with small differences in the list data type processed in a stack like fashion.

HList vs DList In [36] the input list is an heterogeneous list, whose raw types are progressively deconstructed exposing the reified constructor and pushing its children on the stack. In this version instead the `DTree` data type already contains the nodes of the root and its children. Due to the lazy semantics of Haskell the two transformations should have approximately the same performance, however there are several advantages in the encoding proposed in this thesis.

First of all in [36] constructors are compared implicitly, by trying to deconstruct a value of a raw type against the constructor at hand. This is achieved using the partial method `fields` of type, adapted to our definitions, `f as a -> a -> Maybe (HList as)`, which is included in the type class `Diff` and it is used to implement the auxiliary function `matchConstructor`, in the `diff` algorithm. In this version instead such method is requires as a primitive, specifically in the operator `=?=` in the type class `Family` 4.1.2, which explicitly tests for equality between nodes. It is much more concise to test equality at once using the latter, rather than indirectly trying to match the given constructor, with all the possible constructors of a given type. In addition that method requires the additional type class `Type` that provides a method `constructors`, which returns a list of all the constructors of a given type. Secondly comparing the nodes directly does not require to explicitly distinguish between concrete and abstract data types, as it happens in [36]. In summary the encoding provided by `DList` yields a minimal and neat design that requires less boilerplate code, which is often source of nasty bugs.

Patch The `patch` function described in [36] provides the same interface of GNU patch and applies an edit script to the source object, encoded as an heterogeneous list. The semantics of delete and consequently copy relies on the partial function `fields`, which deconstructs the input value expecting a certain specific constructor and fails ungracefully, if this is not the case. As a matter of fact the edit script data type already contains enough information to reconstruct *both* the source and target object. Section 4.2.3

and 3.2.2 show two total function that retrieve both from the edit script alone. The formal model heavily relies on this property and the advantages with respect to safety for a practical implementation are evident. Note that the same could be achieved in [36] using exclusively the insert and apply function.

Modularity The universe representation proposed by Lempsink et al. in [36] handles families of mutually recursive data types. Specifically it requires to collect in a unique data type all the constructors of each type present in the family of mutually recursive data types at hand. This choice of encoding does not support polymorphic data types, such as list, which must be restricted to be monomorphic. As a result the same polymorphic data type, instantiated differently, need separate labels in the representative family, resulting in code duplication. For example the type [[Int]] would need the following universe:

```
data Table xs a where
  Int' :: Int -> Table '[] Int

  INil :: Table '[] [Int]
  ICons :: Table '[Int, [Int]] [Int]

  LNil :: Table '[] [[Int]]
  LCons :: Table '[[Int], [[Int]]] [[Int]]
```

The universe representation is not polymorphic, hence two different witnesses are needed for [] and (:) : INil and ICons for [Int], LNil and LConst for [[Int]].

The encoding proposed in this thesis does not need to handle all the types involved at once, but each instance and representation is given separately per type. This approach support s polymorphic data types and it is modular, allowing code reuse. For example the same example would need only one instance and one representative family for lists:

```
data ListF xs a where
  Nil :: ListF '[] [a]
  Cons :: ListF '[a , [a]] [a]

instance Diff a => Diff [a] where
  type FamilyOf [a] = ListF
```

Given an instance for Diff Int an appropriate instance can be automatically generated for [[Int]], reusing the polymorphic list instance twice. From Diff

Int, the instance Diff [Int] can be derived, which is then used to derive Diff [[Int]].

```
data IntF xs a where
  Int' :: Int -> IntF '[] Int

instance Diff Int where
  type FamilyOf Int = IntF
```

See chapter/appendix 5 for a complete example.

4.3 Diff$_3$

This section shows the Haskell implementation of the diff3 algorithm discussed in 3.3.2. The differences are minimal and the only remarkable distinction is that the implementation is less safe than the one proposed in the formal model, because of the limitations of data type promotion.

4.3.1 Edit Script

The ES3 xs data type represents a merged edit script, whose source object has type list xs and it is almost identical to the ES data type.

```
data ES3 xs where
  Ins3 :: Diff a => F xs a -> ES3 ys -> ES3 ys
  Del3 :: Diff a => F xs a -> ES3 (xs :++: ys) -> ES3 (a ': ys)
  Upd3 :: Diff a => F xs a -> F ys a -> ES3 (xs :++: zs) -> ES3 (a ': zs)
  Cnf3 :: VConflict -> ES3 xs -> ES3 ys
  End3 :: ES3 '[]
```

The edit script is type safe only with respect to the source list for the reasons listed in 3.2.5. It is convenient to retain this bit of type safety because it simplifies the type checking phase, which then has to detect type errors with respect to the output list only. The VConflict data type denotes the presence of a conflict and corresponds to the Conflict data type discussed in 3.2.4.

```
data VConflict where
  InsIns :: (Diff a, Diff b) => F xs a -> F ys b -> VConflict
  UpdDel :: Diff a => F xs a -> F ys a -> VConflict
  DelUpd :: Diff a => F xs a -> F ys a -> VConflict
  UpdUpd :: Diff a => F xs a -> F ys a -> F zs a -> VConflict
```

However since conflicts already produce an ill-typed edit script it has been simplified not to include in its type any information about the values involved.

4.3.2 Algorithm

The function merge3 merges the changes from two aligned edit scripts and corresponds to that presented in 3.3.2.

```
type family (:++:) (xs :: [ k ]) (ys :: [ k ]) :: [ k ] where
  '[] :++: ys = ys
  (x ': xs) :++: ys = x ': (xs :++: ys)
```

Notably Haskell does not allow to promote GADTs [71], therefore it is impossible to define a data type that ensures the alignment of two edit scripts, such as $e_1 \vee e_2$. Consequently the alignment condition is checked at run-time by the function aligned and the merge is aborted ungracefully if this is not the case.

```
data HList (xs :: [ * ]) where
  Nil :: HList '[]
  Cons :: x -> HList xs -> HList (x ': xs)
```

For the same reason the merging conditions, described in the formal model by $f \sqcup g \downarrow h$, are checked directly comparing for equality the nodes involved. The corresponding conditions have been added in form of a comment to help the reader's intuition. The Nop operations have been excluded from the edit script data type and the effect of the extension discussed in 3.3.2 is achieved via pattern matching. In particular the Ins-Ins case is matched first and in the remaining cases, in which only one of the two edits is insert, the insert is just added to the merged script. In the formal model this is equivalent to aligning Ins α with a dummy Nop operation, which would produce the former upon merge.

4.3.3 Type Checking

In order to retrieve the merged target object, the edit script produced by merge3 has to be type checked in order transform it into a well-typed edit script. The algorithm implements the typing rules listed in 3.2.6 and reports all the type errors detected in a script. The following data type encodes the target type list inferred for an edit script.

```
data InferredType xs where
  INil :: InferredType '[]
  ICons :: (x :<: f) => Proxy x -> InferredType xs -> InferredType (x ': xs)
  Top :: InferredType xs
```

Additionally the constructor Top denotes an arbitrary list of types and it is assigned to edit scripts that are ill-typed or that contain a conflict. The data type IES xs pairs together an edit script and its inferred type.

```
data IES xs where
  IES :: InferredType ys -> ES xs ys -> IES xs
```

Note that the list of types ys must be existentially quantified, because it is not possible to know in advance the resulting type, as was explained in 3.2.6.

Type errors Type errors are represented by the data type TypeError:

```
data TypeError where
  TyErr :: ExpectedType xs -> InferredType ys -> TypeError
```

The type InferredType ys denotes the actual type found for some edit script, while the data type ExpectedType xs is just a wrapper around TList xs:

```
newtype ExpectedType xs = ET (TList xs)
```

Type errors and value conflicts are all reported as conflicts while type checking:

```
data Conflict = VConf VConflict
              | TConf TypeError
```

Unification The second typing rule discussed in section 3.2.6 requires to check whether the list of types xs expected as argument by some node of type f xs a is a prefix of zs, the list of output types inferred for the edit script at hand. The data type IsPrefixOf xs zs represents such a proof:

```
data IsPrefixOf xs zs where
  Prefix :: InferredType ys -> Unify (xs :++: ys) zs -> IsPrefixOf xs zs
```

More precisely xs is a prefix of zs, if there is a suffix ys, possibly empty, such that xs :++: ys equals to zs. The type Unify as bs denotes that the two list of types as and bs can be unified.

```
data Unify as bs where
  Same :: Unify as as
  Failed :: Unify as bs
```

The second constructor Failed it is used to handle properly the super type Top, which by assumption unifies with any type.

With these definitions in place it is straightforward to implement the function isPrefixOfTy, which checks whether some concrete list of types is a prefix of the given inferred list:

```
isPrefixOfTy :: TList as -> InferredType bs -> Maybe (IsPrefixOf as bs)
isPrefixOfTy TNil s = Just (Prefix s Same)
isPrefixOfTy s Top = Just (Prefix Top Failed)
isPrefixOfTy (TCons _ _) INil = Nothing
isPrefixOfTy (TCons x s1) (ICons y s2) =
  case (tyEq x y, isPrefixOfTy s1 s2) of
    (Just Refl, Just (Prefix s Same)) -> Just (Prefix s Same)
    (Just Refl, Just (Prefix s Failed)) -> Just (Prefix s Failed)
    _ -> Nothing
```

The function is defined by induction on the two lists. In the first base case it follows immediately that the empty list is the prefix of any list. In the second case, any list unifies with Top by assumption, hence instead of failing the unifier Failed is used. Note that without this constructor the type checker would prevent us to produce a value of type IsPrefixOf. On the other hand in the third base case a nonempty list cannot possibly unify with the empty list, hence Nothing is returned. In the last recursive case, in which both the lists are nonempty, the first types are compared and isPrefixOfTy is called recursively on their tails. If a list is a prefix of another, adding the same type to both the two lists preserves the property. For this reason if the first tails is a prefix of the second and the two types are equal the same suffix s is retained. Note that it is necessary to explicitly pattern match on the unifier to convince the type checker of this property. If these conditions are not met, a negative answer is reported in form of Nothing.

Algorithm Since it is preferable to report at once all the conflicts found in an edit script the type checker reports a list of conflicts and the converted typed edit script, whose type have been inferred.

```
tyCheck :: Family f => ES3 xs -> ([Conflict], IES xs)
tyCheck End3 = ([], IES INil End)
tyCheck (Cnf3 c e) =
  case tyCheck e of
    (tyErr, IES ty e') -> (VConf c : tyErr, IES Top ⊥)
tyCheck (Del3 x e) =
  case tyCheck e of
```

```
    (tyErr, IES ty e') -> (tyErr, IES ty (Del x e'))
tyCheck (Ins3 x e) =
  case tyCheck e of
    (tyErr, IES ty e') ->
      let xs = argsTy x in
      case xs `isPrefixOfTy` ty of
        Just (Prefix xsys Same) -> (tyErr, IES (ICons Proxy xsys) (Ins x e'))
        Just (Prefix xsys Failed) -> (tyErr, IES (ICons Proxy xsys) (Ins x ⊥))
        Nothing -> (TConf (TyErr (ET xs) ty) : tyErr, IES (ICons Proxy Top) (Ins x ⊥))
tyCheck (Upd3 x y e) =
  case tyCheck e of
    (tyErr, IES ty e') ->
      let ys = argsTy y in
      case ys `isPrefixOfTy` ty of
        Just (Prefix yszs Same) -> (tyErr, IES (ICons Proxy yszs) (Upd x y e'))
        Just (Prefix yszs Failed) -> (tyErr, IES (ICons Proxy yszs) (Upd x y ⊥))
        Nothing -> (TConf (TyErr (ET ys) ty) : tyErr, IES (ICons Proxy Top) (Upd x y ⊥))
```

In the base case the empty edit script is converted to the corresponding typed script, whose output type list is empty. The value related conflicts are simply added to the list of conflicts and since there is no edit script counterpart the converted edit script is filled with $\perp$, i.e. undefined, and assigned type Top. In the Del3 case no check is required, because only the input list is affected by this edit and the ES3 data type is already well-typed with respect to it. On the other hand converting the Ins3 x and Upd3 x y edits requires to verify that the output type of the rest of the edit script contain the types demanded respectively by the constructors x and y. These types are extracted using argsTy and the auxiliary function isPrefixOf is used to match them against ty, the type inferred for the rest of the edit script, obtained by a recursive call to tyCheck. Once more pattern matching on the unifier object is required to actually convince the type checker. When the unification is vacuous, denoted by Failed, the converted edit scrip is substituted with $\perp$. Nevertheless, even in these circumstances, part of the correct type is inferred and reported, so that further type errors, independent from the previous ones, can still be detected.

Note that using $\perp$ is safe because the converted edit script is never inspected in tyCheck and thus never evaluated according to Haskell's lazy semantics. The invariant maintained by tyCheck is that the inferred edit script is fully defined only if no conflicts have been detected. Furthermore in that case the inferred type does not contain Top. Therefore a safer interface is provided by means of typeCheck, which returns either a non empty list of conflicts or a well-type edit script.

```
typeCheck :: Family f => ES3 xs -> Either [Conflict] (WES xs)
typeCheck e =
  case tyCheck e of
    ([]  , IES ty e') -> Right $ WES (toTList ty) e'
    (errs, _        ) -> Left errs
```

The data type WES stands for well-typed edit script, and it differs from IES because the output type ys is stored as a TList, instead of InferredType and therefore does not contain any Top type. Note that, even if a script is well-typed, its output type list still needs to be existentially quantified.

```
data ES xs where
  WES :: TList ys -> ES xs ys -> WES xs
```

Finally for user's convenience the canonical diff3 interface is provided, which expects three arguments, the second being the original version and the first and the third being the new ones. Note that those must have the same type, hence in this specific case the expected output type of the edit script is known. In this function the converted edit script is therefore also type-checked against it and an appropriate conflict is reported otherwise.

```
diff3 :: (Diff a, Diff b) => b -> a -> b -> Either [Conflict] (ES '[ a ] '[ b ])
```

4.4 Version Control System

This sections presents a prototype of a structure-aware version control system that put the results presented previously into practice. The prototype is not meant to be of production quality, but rather a proof of concept that shows the applicability of the ideas discussed in this thesis.

4.4.1 Design

The design of the prototype is inspired by Git [11]. The repository is modeled as directed acyclic graph (DAG), in which every node may have at most two parents. The only node without parents is the root, which contains the initial state; nodes with only one parent denote single commits, while nodes with two parents are reserved for merges between two branches. Each node in the graph is identified by the hash of the path that goes from the root to it. Since the most common operations, e.g. commit, branching and merging, are performed on the tip of a branch, paths are stored backwards, so that these operations can be implemented efficiently.

```
data Path a = Root a
            | Node  (Path a) Depth (Delta a)
            | Merge (Path a) (Path a) Depth (Delta a)
```

To keep the prototype simple, objects under revision are restricted to keep the same type a. Except the root, all nodes do not store a plain value, but only a Delta, i.e. an edit script that contains the differences from the previous version.

```
type Delta a = ES '[ a ] '[ a ]
```

In addition each non-root node stores its depth, i.e. the length of the path that goes from the root to it. The depth of a path can be easily retrieved:

```
depth :: Path a -> Depth
depth (Root _) = 0
depth (Node _ d _) = d
depth (Merge _ _ d _) = d
```

Paths have strictly increasing depths, therefore, in order to maintain this invariant, the Path data type is kept abstract and smart constructors are provided instead:

```
root :: a -> Path a
root = Root

node :: Path a -> Delta a -> Path a
node p = Node p (depth p + 1)

merge :: Path a -> Path a -> Delta a -> Path a
merge p1 p2 e = Merge p1 p2 (max (depth p1) (depth p2) + 1) e
```

Furthermore the current value of an object can reconstructed from the path:

```
value :: Diff a => Path a -> a
value (Root x) = x
value (Node _ _ e) = patch e
value (Merge _ _ _ e) = patch e
```

Where patch is a type-restricted version of target which additionally converts a DTree to a raw value. Its implementation is straightforward and thus omitted.

```
patch :: Diff b => ES '[ a ] '[ b ] -> b
```

Since the target object can be retrieved directly from an edit script without the need of the source object, computing the current value of a path does not require patching all the previous deltas from the root.

4.4.2 Lowest Common Ancestor

The lowest common ancestor, henceforth LCA, of two nodes is the lowest node, i.e. deepest from the root, that is an ancestor of both of them. In a tree the LCA is unique, however in a DAG there could be more. In a connected DAG, in which every node has at most two parents, there are at most two lowest common ancestors for every node. This property is useful when merging two branches and it is explained in more detail in 4.4.3. The LCA is represented accordingly by the following data type:

```
data Lca a = One (Path a)
           | Two (Path a) (Path a)
```

Furthermore to take advantage of the faster hash-based comparison for paths, the wrapper `HPath` is used:

```
newtype HPath a = HPath {hpath :: Path a}
```

The appropriate `Eq` and `Ord` instances are assumed for it.

Two auxiliary functions are used to compute the LCA of two paths. Firstly the function `levels` pairs all the subpaths of the given path by their depth in descending order.

```
levels :: Hashable a => Path a -> [(Depth, Set (HPath a))]
levels r@(Root x)= [(0, singleton (HPath r))]
levels n@(Node p d _) = (d, singleton (HPath n)) : levels p
levels m@(Merge p1 p2 d _) = (d, singleton (HPath m)) : combine (levels p1) (levels p2)
```

Secondly the function `combine` merges two such lists, combining the paths at the same depth. Note that due to the invariants discussed previously, the list returned contains an element for each depth level from zero to the depth of the input path.

```
combine :: Hashable a => [(Depth, Set (HPath a))] -> [(Depth, Set (HPath a))]
        -> [(Depth, Set (HPath a))]
combine [] ds2 = ds2
combine ds1 [] = ds1
combine a@((d1,xs) : ds1) b@((d2,ys) : ds2) =
  case compare d1 d2 of
```

```
    LT -> (d2, ys) : combine a ds2
    EQ -> (d1, xs `union` ys) : combine ds1 ds2
    GT -> (d1, xs) : combine ds1 b
```

Finally the function lca computes the lowest common ancestor of two paths.

```
lca :: Hashable a => Path a ->  Path a -> Lca a
lca p1 p2 =
  case find (not . null) (zipWith common ls1 ls2) of
    Just s ->
      case toList s of
        [ r ] -> One (hpath r)
        [r1  r2] -> Two (hpath r1) (hpath r2^^l)
  where d = min (depth p1) (depth p2)
        ls1 = dropWhile ((> d) . fst) (levels p1)
        ls2 = dropWhile ((> d) . fst) (levels p2)
        common (_, x) (_, y) = intersection x y
```

The lists ls1 and ls2 contain the subpaths of each of the two paths, starting with the same common depth d. Since the level d is the minimum depth of the two paths, the lowest common ancestor cannot be at any depth greater than d, hence dropping elements greater than d is safe. Furthermore since any path at a certain depth contains subpaths at every level lower than it, it follows that ls1 and ls2 are aligned with respect to their depth. Exploiting this property the subpaths at each level are combined by intersection, finding thus the common ones. Since these are also in descending order, the first non empty set contains the deepest, i.e. the lowest. The set contains either one or two subpaths, which are then extracted and wrapped in the right constructor. Haskell semantics ensure that the lists of subpath and the list of common paths are produced and consumed lazily, therefore avoiding to unnecessarily process the paths at lower depths.

4.4.3 Merge

The *three-way merge* algorithm merges two branches using the diff3 algorithm, in which the two new versions are the latest in the branches, while their LCA's version is used as base. The LCA represents the best choice because it is a common revision from which both the two branches diverged and furthermore it is the lowest, therefore the closest to both of them. However in some cases such as the *criss-cross merge*, there could be two lowest common ancestors and neither of them is better than the other.

Criss-cross merge The criss-cross merge occurs when there are two separate branches in which each branch progressively includes changes from the other, as shown in the figure 4.1 Depending on the role of the branches and the intended work-flow, this pattern can arise quite often.

For example imagine a repository in which there are two branches, the main branch `master` and a development branch `dev`. The `dev` branch, after adding some feature in commit `2`, is synchronized with master, pulling the changes made in `1`. After solving any possible conflict, the merge is committed in `4`. Similarly the `master` branch is merged with `dev`, adding the new feature from `2` and producing a new commit `3`. In both these merges the lowest common ancestor is the root `0`, which is used as base in the `diff3` algorithm. The pattern repeats itself and now the nodes `3` and `4` must be merged: what node should be used as base? Crucially the nodes `1` and `2` are both LCA, because they are ancestors of both of them and they have the same depth, however none of them is better than the other. Furthermore choosing arbitrarily one of them would raise bogus conflicts, because the changes made in the other would not be taken into account. Choosing an older, but unique ancestor is not satisfactory either. In this case for example using node `0` as the candidate base would raise the same conflicts encountered when merging `1` and `2`, which were already solved.

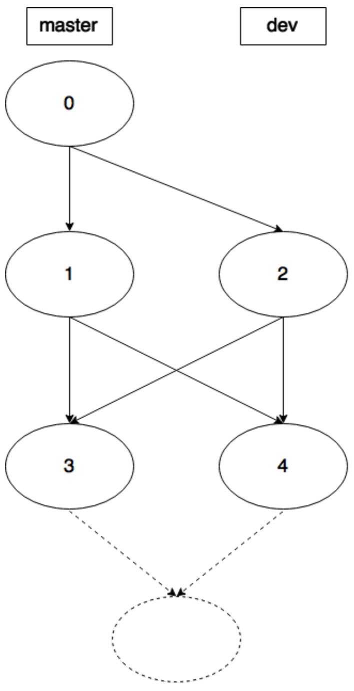

Figure 4.1: Example of criss-cross merge.

4.4.4 Recursive Three-Way Merge

The *recursive three-way merge* algorithm is an extension of the simple three-way merge algorithm, that applies in these circumstances. When two LCA are found, it builds a *virtual* lowest common ancestor applying the three-way merge algorithm recursively on them. The virtual ancestor is then used as a base in the diff3 algorithm. Since the number of nodes in every path are finite the algorithm eventually terminates.

Example In the previous example, the nodes 3 and 4 have two LCA, nodes 1 and 2, therefore the three-way merge is recursively applied to them. Their LCA, node 0, is used as base to build a virtual ancestor 1-2, which is finally used as base to merge the nodes 3 and 4.

The recursive three-way merge can be easily implemented as follows:

```
recursive3WayMerge :: (Hashable a, Diff a) => Path a -> Path a
                      -> Either [Conflict] (Path a)
recursive3WayMerge p q =
  case lca p q of
    One a -> mergeWithAncestor p a q
    Two a b ->
      case recursive3WayMerge a b of
        Left err -> Left err
        Right c -> mergeWithAncestor p c q
```

The function mergeWithAncestor merges two nodes applying the diff3 algorithm. The arguments are expected in the same order as diff3, hence the second Path is used as base.

```
mergeWithAncestor :: (Hashable a, Diff a) => Path a -> Path a -> Path a
                     -> Either [Conflict] (Path a)
mergeWithAncestor p a q =
  case diff3 x o y of
    Left err -> Left err
    Right e -> Right (merge p q e)
  where x = value p
        o = value a
        y = value q
```

4.4.5 Discussion

The prototype presented in this section lacks most of the features commonly expected in version control systems. For example they usually provide (graphical) user interface, record meta-data such as branches names and time and date of commits and so on. They also communicate over the network and serialize data and meta-data for persistence. Part of these features mostly require a great deal of software engineering work and therefore have not been addressed in this project, nevertheless the prototype have all the essential ingredients to perform revision control interactively, including semi-automatic merge of revisions using the *recursive three-way merge* algorithm.

File system Version control systems such as Git [11] and Mercurial [48] do not track single objects, but a collection of files and directories. The prototype discussed in this thesis is in principle able to correctly simulate the same behaviour. In fact it is possible to model the portion of the file system under revision as a mapping from paths to file contents. Providing appropriate support for unordered collections such as maps, it would be possible to instantiate the repository, as `Path (Map FileName Content)`, which would be initialized with the empty mapping. More research is needed to properly handle arbitrary data types that mix structured, tree-liked data types, with unordered collections. The problem does not concern the alignment phase only, but it is also semantic. For instance it is reasonable to expect that two maps are compared, diffing pointwise values mapped by the same key and including the entries that are present in one, but not in the other. However this approach would not handle properly changes made exclusively to the keys, which could happen for example by renaming a file. These questions can dramatically affect the behaviour of a structure-aware version control system and ought to be investigated.

Semantics The model presented in chapter 3 is suitable for reasoning about merging between branches according to the simple three-way merge algorithm. Likewise it would be useful to extend it to consider more complex merges, that require the *recursive* three-way merge. This extension would require to model the repository directly and to formally define the notion of lowest common ancestor. Furthermore the properties that were only informally stated in this section could be formally proved.

Chapter 5

Example

This chapter summaries the contributions of this thesis with a concrete fully worked-out example. It includes a precise definition of the format examined, the code[1] that implements in a unique specification both parser and printer for it and the instances needed to diff and merge values of that format. Examples of successful and unsuccessful merges are given.

5.0.1 Csv

The comma-separated values (CSV) format is used to store tabular data in a file. Each line of text correspond to a row in the table, which is divided in columns by comma characters. In each cell there is a single value, which for simplicity it is assumed to be an integer number.

In Haskell a Csv table can be described as a list rows, each of which is a list of integers:

```
type Row = [Int]
type Csv = [Row]
```

Csv Format The library developed in chapter 2 provides few combinators, that are needed to describe this format.

```
newline :: Format c m Char '[]
newline = char '\n' <?> "If new-line"

sepBy, sepBy1 :: Format c m i xs -> Format c m i '[] -> Format c m i (Map [] xs)
```

[1]In order to keep the code readable the *closed* format representation is used and class constraints are omitted from signatures.

The formats sepBy and sepBy1 are standard combinators implemented in terms of many. The format sepBy f s repeatedly recognize the format f zero or more times, separated by the format recognized by s. Similarly sepBy1 f s succeeds only if f is recognized at least once.

Furthermore an appropriate integer format is assumed:

```
int :: Format c m Char '[Int]
```

With these pieces in place it is straightforward to describe the Csv format as:

```
csvFormat :: Format c m Char '[ Csv ]
csvFormat = sepBy row newline
  where row = sepBy1 int (char ',')
```

Choosing an appropriate parsing and printing backend, a csv parser and printer are derived from csvFormat.

```
csvParser :: Parser Csv
csvParser = hHead <$> mkParser csvFormat

csvPrinter :: Csv -> Maybe String
csvPrinter = mkPrinter csvFormat . hsingleton
```

Csv Diff In order to concretely show the benefits of a structure-aware algorithm for diffing and merging data, the unsatisfactory failure of GNU diff is reported.

Imagine that a repository stores the following table:

1	2	3
4	5	6
7	8	9

Which is stored in a CSV file:

```
$ cat o.csv
1,2,3
4,5,6
7,8,9
```

Now suppose that two users, Alice and Bob, make the following changes:

```
alice$ cat o.csv
0,1,2,3
0,4,5,6
0,7,8,9
```

```
bob$ cat o.csv
1,2,3
4,5,9
7,8,15
```

A line-based merging algorithm such as GNU diff3 would report a conflict in the second and third line, because Alice, Bob and the original version are all different. The first line instead is merged correctly because only Alice changed it.

Alice	Original	Bob	Merged
0,1,2,3	1,2,3	1,2,3	0,1,2,3
0,4,5,6	4,5,6	4,5,9	✗
0,7,8,9	7,8,9	7,8,15	✗

Using a structure aware algorithm the two tables can be successfully merged, producing the following table:

0	1	2	3
0	4	5	9
0	7	8	15

In order to merge the Csv data type, an appropriate instance for Diff is needed. The two data types involved in Csv are lists and integers. For both of them the library Typable already provide a suitable instance, hence the superclass constraint of Diff is satisfied.

First of all we need to define a representative family for the constructors of lists.

```
data ListF xs a where
  Nil :: ListF '[] [a]
  Cons :: ListF '[a, [a]] [a]
```

In the corresponding Diff instance, the type ListF is assigned to the associated type FamilyOf. The implementation of the other methods is straightforward:

```
instance Diff a => Diff [a] where
  type FamilyOf [a] = ListF

  Nil =?= Nil = Just Refl
  Cons =?= Cons = Just Refl
  _ =?= _ = Nothing

  distance Nil Nil = 0
  distance Cons Cons = 0
  distance _ _ = 1

  fromDTree (Node Nil DNil) = []
  fromDTree (Node Cons (DCons x (DCons xs DNil))) = fromDTree x : fromDTree xs
  toDTree [] = Node Nil DNil
  toDTree (x:xs) = Node Cons ds
    where ds = DCons (toDTree x) $ DCons (toDTree xs) DNil

  argsTy Nil = tlist
  argsTy Cons = tlist

  string Nil = "[]"
  string Cons = "(:)"
```

In the methods argsTy, after patter matching on the witness F xs a, the arguments get into scope, and hence it is possible to automatically build the required TList. Note that to fulfill this method and the embedding fromDTree and toDTree, an instance of Diff a is needed, therefore its presence in the instance context is not optional.

Similarly a representative family is defined for integers. Basic types and abstract data types can be represented wrapping their value in a dummy constructor.

```
data IntF xs a where
  Int' :: Int -> IntF '[] Int
```

Likewise an appropriate Diff instance is defined for Int:

```
instance Diff Int where
  type FamilyOf Int = IntF

  (Int' x) =?= (Int' y) = if x == y then Just Refl else Nothing
  distance (Int' x) (Int' y) = if x == y then 0 else 1

  fromDTree (Node (Int' n) DNil) = n
  toDTree n = Node (Int' n) DNil
  argsTy (Int' _) = TNil
  string (Int' i) = show i
```

Note that for consistency we have to compare the integer also in the method =?=.

Suppose that the previous files have been successfully parsed using csvParser and the raw csv table extracted:

```
c0, c1, c2 :: Csv
c0 = [[1,2,3],[4,5,6],[7,8,9]]
c1 = [[0,1,2,3],[0,4,5,6],[0,7,8,9]]
c2 = [[1,2,3], [4,5,9], [7,8,15]]
```

The library developed in chapter 4 includes an auxiliary function that reconstructs the merged value when no conflicts are raised:

```
diff3Patch :: :: (Diff a, Diff b) => b -> a -> b -> Either [Conflict] b
diff3Patch x o y =
  case diff3 x o y of
    Left errs -> Left errs
    Right e -> Right (patch e)
```

The two tables are successfully merged and produce the expected result:

```
*> diff3Patch c1 c0 c2
Right [[0,1,2,3],[0,4,5,9],[0,7,8,15]]
```

The merged table is extracted to `c012` and printed back according to its format specification:

```
*> putStrLn (printCsv c012)
0,1,2,3
0,4,5,9
0,7,8,15
```

In order to show a negative example, consider the following table:

```
*> diff3 c2 c0 c3
Left [VConf UpdUpd 6 9 18,VConf UpdUpd 9 15 30]
```

Merging `c2` with `c3`, keeping as base `c0`, fails with the following conflicts:

```
*> diff3 c2 c0 c3
Left [VConf UpdUpd 6 9 18,VConf UpdUpd 9 15 30]
```

These are true conflicts: the sixth element of `c0` (`6`) has been updated to `9` in `c2` and to `18` in `c3`. Likewise the ninth element of `c0` (`9`) has been changed with `15` in `c2` and with `30` in `c3`. These changes are incompatible and therefore two conflicts are correctly reported.

Chapter 6

Conclusion

In this thesis I have analyzed the problem of structure-aware revision control, which aims to improve the quality of version control systems, by exploiting the knowledge of how data is encoded in a file.

Firstly I have developed an EDSL for binary and text-based data formats, which automatically derives inverse-by-construction parser and printer, given a format description. Since a unique specification is given for each format, this technique ensures that parsers and printers are always synchronized, therefore ensuring round-trip behaviour. A format description allows a version control system to access to the data contained in a file and its structure, so that it can detect changes more precisely. In addition after a merge, the data can be serialized back to a file according to its format.

Secondly I have implemented a data-type generic `diff` and `diff3` algorithm in Haskell. Version control systems employ these algorithms to perform their basic operations: `diff` tracks the history of changes made to some data, while `diff3` merges separate revisions. Generic programming techniques are required, because it is unfeasible to implement specific versions of these algorithms, due to the multitude of different data formats available. The data stored in a file is parsed producing a domain-specific data type, which is then converted to the heterogeneous rose trees data type, the generic representation employed in the `diff` and `diff3` algorithms. The algorithms have been embedded in a proof-of-concept structure-aware version control system that shows the applicability of the theories studied in this thesis.

Lastly I have developed a formal model in the Agda proof assistant, with which I have studied the two algorithms and their formal properties. The model provides an unambiguous specification of the algorithms and accurately describe their semantics. In addition the properties proved are a precious source of information, useful to interpret and further describe their behaviour.

The contributions of this thesis can be summarized in:

- An EDSL for describing data formats that unifies parsing and printing.
- A data type generic `diff` and `diff3` algorithm.
- A formal model that describes the semantics of these algorithms.
- A proof-of-concept structure-aware version control system.

6.0.1 Related and Future Work

This thesis covers several related topics: invertible parsing and printing, change detection and automatic merging of structured data and semantics of version control systems and synchronizers. This is the first work of my knowledge that has combined results from these studies in the development of a proof-of-concept structure-aware version control system, that employs data-type generic `diff` and `diff3` algorithms, whose semantics has been formalized in a proof assistant.

Semantics of Format Description Chapter 2 presents a promising approach to parsing and printing unification, which aims to derive inverse-by-construction parser and printer for a given format, using a format combinator library inspired by [56]. The central idea of the library is to define a small core of basic combinators, that essentially correspond to those of the `Functor`, `Applicative` and `Alternative` classes, which embody both the parsing and printing semantics at once. Complex formats are described combining the basic combinators just like it happens in parser combinator libraries. The library assumes that the double semantics of each core combinator is *valid*, i.e. each one inverts the other, and conjectures that parsers and printer obtained by composing them will be therefore *consistent*. However it is a fundamental open research question whether the composition of *valid* formats does in fact produce a *valid* format. Section 2.2.6 already hinders this conjecture, since the inverse printer of a simple parser, was actually non terminating. Nevertheless termination is usually a thorny problem alone and in this setting it is further complicated by the mix of inductive and coinductive definitions, typical of parser combinator libraries. Danielsson develops a library of total parser combinators [16], that ensure termination exploiting dependent types [16]. More specifically both terminating parser and printer could be described introducing additional indexes that respectively mark infinite from finite parts of parsers and printers. His work certainly represents an excellent starting point to tackle the problem. This technique also forms the basis for another work on correct-by-construction pretty printers [17]. Pretty-printers are indexed by a grammar and their output is valid with respect to it. It seems possible to combine these promising results to formally prove the correctness of a format combinator library based on these ideas.

Move Edit The set of edits available to describe the changes between two revisions is a crucial factor in the automatic merging capabilities of a version control system. The more precise an edit script is, the less likely it is to trigger a false conflict when merging. Rearrangements of portions of texts occur often in certain data formats. For instance in a web-page, sections may be moved around, or in a software artifact a refactoring tool may reorder the list of methods in alphabetical order. All these changes are currently detected by GNU diff and the diff presented in this thesis as a series of inserts followed by deletes, which is just a rough approximation. Other tools, like Lindholm's 3DM [37], provide a move edit, which appropriately represent these changes. It would be interesting to include a move edit in the edit data type and study how contexts [26, 43] can improve the alignment phase.

Semantics of Version Control The formal model presented in this thesis specifically addresses the semantics of the data-type generic diff and diff3 algorithms employed in a structure-aware version control system. As such it gives the foundations to further investigate the semantics of structure-aware version control systems. Swiestra and Löh study the semantics of version control systems using separation logic in which operations are modeled using *Hoare triples* [65]. The operations are applied on the internal model of the repository, a shared stated described with the the preconditions and postconditions predicates, which represent respectively the repository before and after each operation. The history of a repository consists in a valid sequence of operations and branching is modeled with a conditional statement. With their model they intend to give sensible high-level specifications of a version control system, therefore they refrain from giving any specific algorithm for change-detection and for merging branches. I believe it would be interesting to embed the formal model developed in this thesis into a similar more general framework. In particular the framework should explicitly represent branches and nodes, in order to compute the lowest common ancestor that act as a base in the three-way merge.

Swiestra and Löh begin their description considering firstly a single binary file repository, then extend it to text files and lastly consider multiple files and directories. For example they give a formal description of finite mappings that encodes the file-system, the relative operations and their semantics. Clearly the model becomes a little bit more involved with every extension. The advantage of the data-type generic approach discussed in this thesis is that these modifications are superfluous, as long as these data structures can be transformed into the generic representation. Unfortunately the algorithms proposed in this thesis work best with strongly structured data and have poor performances with unordered collections, such as dictionaries. Properly diffing and merging unordered collections is an interesting research question, with important practical ramifications that should be investigated in future work. I expect that a complete satisfactory semantics cannot be given generically, however sets and dictionaries are probably the fundamental unordered collections, that are widespread in data formats. It is worth pointing out that the complication with respect to this class of data structures lies in the *alignment phase* and not in the merging semantics. Currently the alignment is simplified by the fact that trees are processed in depth first order. Equivalent unordered collections can have multiple concrete representations, therefore this approach would misalign nodes and most likely produce bogus conflicts.

Acknowledgments

I would like to thank my supervisor Dr. Wouter Swiestra for his guidance and constant support throughout this project. Whenever I was stuck, he could set me on the right track to a solution, clarifying my problems and putting them in right words. My gratitude goes also to the Software Technology group, whose professors have inspired me and taught me so much in these two years. Their enthusiasm has sparkled in me the interest for functional programming and programming languages technologies to the point that I decided to pursue a PhD in these topics. I also gladly remember the afternoons spent with the Software Technology reading club, during which we were busy getting the most out of interesting papers and in constructive discussions.

I would like to thank from the bottom of my heart my parents for supporting me throughout my studies. None of this would have been possible without you. Studying at Utrecht University is a great opportunity and my only regret is that it would not let me see you, my sister Chiara, my hopefully future brother-in-law Mattia and my grandmother Clara as often as I wished. A special thanks goes to Laura for putting up with me in the busiest moments and sharing with me this fantastic experience. There a re so many great adventures we have lived in these two years and thanks to you there is plenty of pictures to remember them all. I would like to thank my friends from Utrecht: João Pizani, Philipp, Wout and João Alpuim. I really had a great time with all of you and I hope we will keep in touch in the next years. I would like to thank also my friends from Delft: Enrico, Gherardo, Marco, Oana and Bronius. You are wonderful friends and I am very lucky to have met you.

Lastly a special thanks goes to the Haskell Cafe and Agda mailing list for their concrete help, without which I could have not completed this project. These communities have provided me with very interesting insights and answered to my very technical questions.

Bibliography

[1] Artem Alimarine, Sjaak Smetsers, Arjen van Weelden, Marko van Eekelen, and Rinus Plasmeijer. There and back again: Arrows for invertible programming. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, Haskell '05, pages 86–97, New York, NY, USA, 2005. ACM.

[2] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. Semistructured merge: Rethinking merge in revision control systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 190–200, New York, NY, USA, 2011. ACM.

[3] Kenichi Asai. On typing delimited continuations: Three new solutions to the printf problem. *Higher Order Symbol. Comput.*, 22(3):275–291, September 2009.

[4] S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer? In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, MobiCom '98, pages 98–108, New York, NY, USA, 1998. ACM.

[5] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, SPIRE '00, pages 39–, Washington, DC, USA, 2000. IEEE Computer Society.

[6] Richard Bird and Oege de Moor. *Algebra of Programming.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[7] Boomerang- library for invertible parsing and printing. https://hackage.haskell.org/package/boomerang.

[8] Ana Bove. Another look at function domains. *Electronic Notes in Theoretical Computer Science*, 249:61 – 74, 2009. Proceedings of the

25th Conference on Mathematical Foundations of Programming Semantics (MFPS 2009).

[9] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda — a functional language with dependent types. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 73–78, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, September 2009.

[11] Scott Chacon. *Pro Git*. Apress, Berkely, CA, USA, 1st edition, 2009.

[12] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. *SIGPLAN Not.*, 40(9):241–253, September 2005.

[13] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *In POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13. ACM Press, 2005.

[14] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. *SIGMOD Rec.*, 26(2):26–37, June 1997.

[15] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, 25(2):493–504, June 1996.

[16] Nils Anders Danielsson. Total parser combinators. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 285–296, New York, NY, USA, 2010. ACM.

[17] Nils Anders Danielsson. Correct-by-construction pretty-printing. In *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming*, DTP '13, pages 1–12, New York, NY, USA, 2013. ACM.

[18] Olivier Danvy. Functional unparsing. *J. Funct. Program.*, 8(6):621–625, November 1998.

[19] Dominique Devriese and Frank Piessens. On the bright side of type classes: Instance arguments in agda. *SIGPLAN Not.*, 46(9):143–155, September 2011.

[20] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. *SIGPLAN Not.*, 47(12):117–130, September 2012.

[21] Jeroen Fokker. Functional parsers. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 1–23. Springer Berlin Heidelberg, 1995.

[22] J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. Exploiting schemas in data synchronization. *J. Comput. Syst. Sci.*, 73(4):669–689, June 2007.

[23] Glasgow Haskell Compiler - ghc. https://www.haskell.org/ghc/.

[24] Happy - the parser generator for haskell. https://www.haskell.org/happy/.

[25] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *In Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL-III*, pages 1–55. ACM Press, 2007.

[26] Gérard Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, September 1997.

[27] John Hughes. The design of a pretty-printing library. In *Advanced Functional Programming*, pages 53–96. Springer Verlag, 1995.

[28] Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Sci. Comput. Program.*, 43(1):35–75, April 2002.

[29] C. Stephen Johnson. “yacc—yet another compiler compiler”. Technical report, NJ: Bell Telephone Laboratories, 1975.

[30] Sanjeev Khanna, Keshav Kunal, and BenjaminC. Pierce. A formal investigation of diff3. In V. Arvind and Sanjiva Prasad, editors, *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, volume 4855 of *Lecture Notes in Computer Science*, pages 485–496. Springer Berlin Heidelberg, 2007.

[31] Oleg Kiselyov. Strongly typed heterogeneous collections. In *In Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[32] Oleg Kiselyov. Typed tagless final interpreters. In Jeremy Gibbons, editor, *Generic and Indexed Programming*, volume 7470 of *Lecture Notes in Computer Science*, pages 130–174. Springer Berlin Heidelberg, 2012.

[33] Oleg Kiselyov, Simon Peyton, and Jones Chung chieh Shan. Fun with type functions version 2, 2009.

[34] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. *SIGPLAN Not.*, 38(3):26–37, January 2003.

[35] Daan Leijen. Parsec, a fast combinator parser. Technical report uu-cs-2001-35, Institute of Information and Computing Sciences, UtrechtUniversity, 2001.

[36] Eelco Lempsink, Sean Leather, and Andres Löh. Type-safe diff for families of datatypes. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming*, WGP '09, pages 61–72, New York, NY, USA, 2009. ACM.

[37] Tancred Lindholm. A three-way merge for xml documents. In *Proceedings of the 2004 ACM Symposium on Document Engineering*, DocEng '04, pages 1–10, New York, NY, USA, 2004. ACM.

[38] Tancred Lindholm and Torsten Rueger. A fault-tolerant three-way merge for xml and html, 2005.

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

[40] Simon Marlow. Haskell 2010 language report.

[41] Boespflug Mathieu. Functional pearl: Replaying the stack for parsing and pretty printing. http://www.cs.mcgill.ca/~mboes/papers/cassette.pdf, September 2012.

[42] Kazutaka Matsuda and Meng Wang. Flippr: A prettier invertible printing system. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 101–120, Berlin, Heidelberg, 2013. Springer-Verlag.

[43] Conor Mcbride. The derivative of a regular type is its type of one-hole contexts (extended abstract), 2001.

[44] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.

[45] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, May 2002.

[46] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[47] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, pages 230–266, Berlin, Heidelberg, 2009. Springer-Verlag.

[48] Bryan O'Sullivan. *Mercurial - The Definitive Guide: Modern Software for Collaboration.* O'Reilly, 2009.

[49] Nicolas Oury and Wouter Swierstra. The power of pi. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 39–50, New York, NY, USA, 2008. ACM.

[50] Luuk Peters. Change detection in xml trees: a survey.

[51] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, September 2006.

[52] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, Computer and Information Science Department, Levine Hall, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104-6389, July 2004.

[53] Benjamin C. Pierce, Alan Schmitt, and Michael B. Greenwald. Bringing Harmony to optimism: A synchronization framework for heterogeneous tree-structured data. Technical Report MS-CIS-03-42, University of Pennsylvania, 2003. Superseded by MS-CIS-05-02.

[54] Benjamin C. Pierce and Jérôme Vouillon. What's in Unison? A formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.

[55] Norman Ramsey and Elöd Csirmaz. An algebraic approach to file synchronization. *SIGSOFT Softw. Eng. Notes*, 26(5):175–185, September 2001.

[56] Tillmann Rendel and Klaus Ostermann. Invertible syntax descriptions: Unifying parsing and pretty printing. *SIGPLAN Not.*, 45(11):1–12, September 2010.

[57] David Roundy. Darcs: Distributed version management in haskell. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, Haskell '05, pages 1–4, New York, NY, USA, 2005. ACM.

[58] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. *SIGPLAN Not.*, 43(9):51–62, September 2008.

[59] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for gadts. *SIGPLAN Not.*, 44(9):341–352, August 2009.

[60] Tom Schrijvers, Martin Sulzmann, Simon Peyton Jones, and Manuel Chakravarty. Towards open type functions for Haskell. In O. Chitil, editor, *Implementation and Application of Functional Languages,*, pages 233–251. Computing Laboratory, University of Kent, 2007.

[61] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002.

[62] S. doaitse Swierstra and Olaf Chitil. Linear, bounded, functional pretty-printing. *J. Funct. Program.*, 19(1):1–16, January 2009.

[63] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming, Second International School-Tutorial Text*, pages 184–207, London, UK, UK, 1996. Springer-Verlag.

[64] Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008.

[65] Wouter Swierstra and Andres Löh. The semantics of version control. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming 38; Software*, Onward! '14, pages 43–54, New York, NY, USA, 2014. ACM.

[66] Mervi Ranta Tancred Lindholm. A 3-way merging algorithm for synchronizing ordered trees - the 3dm merging and differencing tool for xml, 2001.

[67] P. Wadler and et al. The expression problem. Discussion on the Java-Genericity mailing list, December 1998.

[68] Philip Wadler. A prettier printer. In *Journal of Functional Programming*, pages 223–244. Palgrave Macmillan, 1998.

[69] Philip Wadler. Propositions as types, 2014.

[70] Bernhard Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, SCM '91, pages 68–79, New York, NY, USA, 1991. ACM.

[71] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM.