Structure Aware Version Control

Victor C. Miraldo and Wouter Swierstra

University of Utrecht

13th of October, 2016

The Problem

Imagine Prof. Pink keeps track of his students mark in a CSV file. She finds a mistake in *Alice*'s grade and corrects it. At the same time, Prof. Green decides it is a good idea to add a new column to the CSV file, in order to track surnames.

The Problem

Imagine Prof. Pink keeps track of his students mark in a CSV file. She finds a mistake in *Alice*'s grade and corrects it. At the same time, Prof. Green decides it is a good idea to add a new column to the CSV file, in order to track surnames.

Name	,	*Surname*	,	*Number*	,	*Grade*
Alice	,	*Lane*	,	440	,	7.5
Bob	,	*Wright*	,	593	,	6.5
Carroll	,	*Clark*	,	168	,	8.5

A line-based diff will recognize this as a conflict, where it is clearly not.

Introduction

Version Control, today

- There are lots of tools, with all sorts of different interfaces.
- The majority of these tools use a line-based diff algorithm, which are good in keeping track of changes in some situations, but bad in merging changes together.
- Hence, merging changes almost always requires human interaction.
- Programmers spent a lot of time solving unnecessary conflicts.

Structure of the Presentation

- We will start by giving an informal specification of what *diffing* should consist in.

Structure of the Presentation

- We will start by giving an informal specification of what *diffing* should consist in.
- We proceed to show how could one diff lists according to the LCS.

Structure of the Presentation

- We will start by giving an informal specification of what *diffing* should consist in.
- We proceed to show how could one diff lists according to the LCS.
- From lists we go to Binary Trees and towards a generalization.

Structure of the Presentation

- We will start by giving an informal specification of what *diffing* should consist in.
- We proceed to show how could one diff lists according to the LCS.
- From lists we go to Binary Trees and towards a generalization.
- We (briefly) mention how we implemented these algorithms for the universe of Context Free types.

Structure of the Presentation

- We will start by giving an informal specification of what *diffing* should consist in.
- We proceed to show how could one diff lists according to the LCS.
- From lists we go to Binary Trees and towards a generalization.
- We (briefly) mention how we implemented these algorithms for the universe of Context Free types.
- We then point to a major problem in our implementation. And address some possible fixes.

Diffing

An intuitive specification

Intuitively, a $\mathit{Patch}\ A$ is an object that describes certain changes that when applied to certain values $a : A$ will produce other values of type A.

- Given two values $a_1\ a_2 : A$, we expect to be able to compute a patch $\mathit{diff}\ a_1\ a_2 : \mathit{Patch}\ A$.
- Given one value $a : A$ and a patch $p : \mathit{Patch}\ A$, we expect to be able to apply it and maybe get another element of type A, hence $\mathit{apply}\ p\ a : \mathit{Maybe}\ A$.

The Trivial Diff

According to our specification, we can already define a trivial diff:

```
Patch : Set → Set
Patch A = A × A
diff : {A : Set} → A → A → Patch A
diff x y = (x , y)
apply : {A : Set} ⦃ x : Eq A ⦄
        → Patch A → A → Maybe A
apply ⦃ cmp ⦄ p x
    = if cmp (fst p) x
      then just (snd p)
      else nothing
```

Diffing

A Better Specification

We need a better specification!

- A *Patch* A should describe the *minimal* transformation between two elements of type A.
- We must have efficient algorithms for creating and applying patches.

The previous trivial implementation keeps too much information.

We will use A's structure to avoid storing duplicate information.

Diffing Lists

Assuming we have a *Patch*, *diff* and *apply* for type A; Take:

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

Diffing lists has been well studied. It is exactly the Longest Common Subsequence (LCS) problem [3].

Diffing Lists

Assuming we have a *Patch*, *diff* and *apply* for type *A*; Take:

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

Diffing lists has been well studied. It is exactly the Longest Common Subsequence (LCS) problem [3].

The edit operations we can make in a list are:

```
data PatchList (A : Set) : Set where
  Nil  : PatchList A
  Ins  : A       → PatchList A → PatchList A
  Del  : A       → PatchList A → PatchList A
  Mod  : Patch A → PatchList A → PatchList A
```

The algorithm

Computing an element of *PatchList* *A* given two *List* *A* is not difficult:

$$
\begin{array}{lllll}
\mathit{diff} & : \mathit{List}\ A & \rightarrow\ \mathit{List}\ A & \rightarrow & \mathit{PatchList}\ A \\
\mathit{diff}\ [] & & [] & = & \mathit{Nil} \\
\mathit{diff}\ (x :: xs) & & [] & = & \mathit{Del}\ x\ (\mathit{diff}\ xs\ []) \\
\mathit{diff}\ [] & & (y :: ys) & = & \mathit{Ins}\ y\ (\mathit{diff}\ []\ ys)
\end{array}
$$

The algorithm

Computing an element of *PatchList A* given two *List A* is not difficult:

```
diff : List A → List A → PatchList A
diff []        []        = Nil
diff (x :: xs) []        = Del x (diff xs [])
diff []        (y :: ys) = Ins y (diff [] ys)
diff (x :: xs) (y :: ys)
   = let d1 = Del x                  (diff xs (y :: ys))
         d2 = Ins y                  (diff (x :: xs) ys)
         d3 = Mod (diff x y) (diff xs ys)
     in choose (d1 :: d2 :: d3 :: [])
```

The algorithm

Computing an element of *PatchList* *A* given two *List* *A* is not difficult:

```
diff : List A → List A → PatchList A
diff []        []        = Nil
diff (x :: xs) []        = Del x (diff xs [])
diff []        (y :: ys) = Ins y (diff [] ys)
diff (x :: xs) (y :: ys)
    = let d1 = Del x                  (diff xs (y :: ys))
          d2 = Ins y                  (diff (x :: xs) ys)
          d3 = Mod (diff x y) (diff xs ys)
      in choose (d1 :: d2 :: d3 :: [])
```

Selecting a patch in the non-trivial case is not so straight forward. This notion will be clarified later, with a better example.

Let's run it!

Let's expand the call tree for *diff* [*tt* , *ff*] [*ff* , *ff*]

$$\begin{tikzcd}
 & & \mathit{diff}\ []\ [\mathit{ff}\ ,\ \mathit{ff}] \\
 & \mathit{diff}\ [\mathit{ff}]\ [\mathit{ff}\ ,\ \mathit{ff}] \arrow[ru, "\mathit{Del}"] \arrow[r, "\mathit{Mod}"] \arrow[rd, "\mathit{Ins}"] & \mathit{diff}\ []\ [\mathit{ff}] \\
 & & \mathit{diff}\ [\mathit{ff}]\ [\mathit{ff}] \\
 & & \mathit{diff}\ []\ [\mathit{ff}] \\
\mathit{diff}\ [\mathit{tt}\ ,\ \mathit{ff}]\ [\mathit{ff}\ ,\ \mathit{ff}] \arrow[ruuu, "\mathit{Del}"] \arrow[r, "\mathit{Mod}"] \arrow[rddd, "\mathit{Ins}"] & \mathit{diff}\ [\mathit{ff}]\ [\mathit{ff}] \arrow[ru, "\mathit{Del}"] \arrow[r, "\mathit{Mod}"] \arrow[rd, "\mathit{Ins}"] & \mathit{diff}\ []\ [] \\
 & & \mathit{diff}\ [\mathit{ff}]\ [] \\
 & & \mathit{diff}\ [\mathit{ff}]\ [\mathit{ff}] \\
 & \mathit{diff}\ [\mathit{tt}\ ,\ \mathit{ff}]\ [\mathit{ff}] \arrow[ru, "\mathit{Del}"] \arrow[r, "\mathit{Mod}"] \arrow[rd, "\mathit{Ins}"] & \mathit{diff}\ [\mathit{ff}]\ [] \\
 & & \mathit{diff}\ [\mathit{tt}, \mathit{ff}]\ []
\end{tikzcd}$$

Diffing Binary Trees

Assuming we have a *Patch*, *diff* and *apply* for type *Maybe* *A*, let us now look at how one would define diffing for binary trees of *A*'s.

```
data Tree (A : Set) where
  Leaf  : Tree A
  Node  : A → Tree A → Tree A → Tree A
```

Diffing Binary Trees

Assuming we have a *Patch*, *diff* and *apply* for type *Maybe* A, let us now look at how one would define diffing for binary trees of A's.

```
data Tree (A : Set) where
  Leaf : Tree A
  Node : A → Tree A → Tree A → Tree A
```

Well, although slightly more complicated than *Lists*, *Tree*s are also the least-fixpoint of a functor!

$$\begin{aligned} &\mathit{TreeF} : \mathit{Set} \to \mathit{Set} \to \mathit{Set} \\ &\mathit{TreeF}\ A\ X = \mathit{Maybe}\ (A \times (X \times X)) \end{aligned}$$

$$\begin{aligned} &\mathit{Fix} : (\mathit{Set} \to \mathit{Set}) \to \mathit{Set} \\ &\mathit{Fix}\ F = F\ (\mathit{Fix}\ F) \end{aligned}$$

$$\mathit{Tree}\ A \approx \mathit{Fix}\ (\mathit{TreeF}\ A)$$

Binary Trees as (least) Fixpoints

The important detail here is that $\mathit{Tree}\ A$ is, in fact, the least fixpoint of $\mathit{TreeF}\ A\ X$!

It is not hard to see that $\mathit{TreeF}\ A\ \mathit{1} \approx \mathit{Maybe}\ A$, we call this the *head* of the tree. Hence, we can always represent a $\mathit{Tree}\ A$ by $\mathit{List}\ (\mathit{TreeF}\ A\ \mathit{1}) \approx \mathit{List}\ (\mathit{Maybe}\ A)$.

Binary Trees as (least) Fixpoints

The important detail here is that $\mathit{Tree}\ A$ is, in fact, the least fixpoint of $\mathit{TreeF}\ A\ X$!

It is not hard to see that $\mathit{TreeF}\ A\ 1 \approx \mathit{Maybe}\ A$, we call this the *head* of the tree. Hence, we can always represent a $\mathit{Tree}\ A$ by $\mathit{List}\ (\mathit{TreeF}\ A\ 1) \approx \mathit{List}\ (\mathit{Maybe}\ A)$.

$$
\begin{array}{lll}
\mathit{hd} : \mathit{Tree}\ A \to & \mathit{Maybe}\ A \\
\mathit{hd}\ \mathit{Leaf} & = \mathit{nothing} \\
\mathit{hd}\ (\mathit{Node}\ x\ \_\ \_) & = \mathit{just}\ x
\end{array}
\qquad
\begin{array}{lll}
\mathit{ch} : \mathit{Tree}\ a \to & \mathit{List}\ (\mathit{Tree}\ a) \\
\mathit{ch}\ \mathit{Leaf} & = [] \\
\mathit{ch}\ (\mathit{Node}\ \_\ l\ r) & = l :: (r :: [])
\end{array}
$$

$$
\begin{array}{l}
\mathit{serialize} :: \mathit{Tree}\ A \to \mathit{List}\ (\mathit{Maybe}\ A) \\
\mathit{serialize}\ x = \mathit{hd}\ x :: \mathit{concat}\ (\mathit{map}\ \mathit{serialize}\ (\mathit{ch}\ x))
\end{array}
$$

Diffing Serialized Trees

Now, the question becomes: which transformations a list (of *heads* of a fixpoint) can undergo?

Well, we can just borrow the definition for diff of *List*s, following the lines of Lempsink's [1] work.

Diffing Serialized Trees

Now, the question becomes: which transformations a list (of *heads* of a fixpoint) can undergo?

Well, we can just borrow the definition for diff of *List*s, following the lines of Lempsink's [1] work.

```
data PatchTree (A : Set) : Set where
  Nil  : PatchTree A
  Ins  : Maybe A → PatchTree A → PatchTree A
  Del  : Maybe A → PatchTree A → PatchTree A
  Mod  : Patch (Maybe A)
       → PatchTree A → PatchTree A
```

Diffing Binary Trees

Since *ch* produces a list of trees, a small adaptation to the previous algorithm is needed:

$$\begin{array}{lll}
\mathit{diff} : \mathit{List}\ (\mathit{Tree}\ A) & \to\ \mathit{List}\ (\mathit{Tree}\ a) & \to\ \mathit{PatchTree}\ A \\
\mathit{diff}\ []\quad\ \ [] & & =\ \mathit{Nil} \\
\mathit{diff}\ (x :: xs)\ [] & & =\ \mathit{Del}\ (\mathit{hd}\ x)\ (\mathit{diff}\ (\mathit{ch}\ x +\!\!+\ xs)\ []) \\
\mathit{diff}\ []\quad\ \ (y :: ys) & & =\ \mathit{Ins}\ \ (\mathit{hd}\ y)\ (\mathit{diff}\ []\ (\mathit{ch}\ y +\!\!+\ ys))
\end{array}$$

Diffing Binary Trees

Since *ch* produces a list of trees, a small adaptation to the previous algorithm is needed:

```
diff  : List (Tree A) → List (Tree a) → PatchTree A
diff []       []       = Nil
diff (x :: xs) []      = Del (hd x) (diff (ch x ++ xs) [])
diff []       (y :: ys) = Ins (hd y) (diff [] (ch y ++ ys))
diff (x :: xs) (y :: ys)
    = let d1 = Del  (hd x) (diff (ch x ++ xs) (y :: ys))
          d2 = Ins  (hd y) (diff (x :: xs) (ch y ++ ys))
          d3 = Mod  (diff (hd x) (hd y))
                    (diff (ch x ++ xs) (ch y ++ ys))
      in choose (d1 :: d2 :: d3 :: [])
```

The notion of Cost

Focusing on the last case of the diff function, we find we have to choose between three patches: $\mathit{choose}\ (d1 :: d2 :: d3 :: [])$.

Our specification mentions we want our patches to be *minimal*.

Our notion of *minimality* is expressed by the means of a cost function,

$$\mathit{cost} : \mathit{Patch}\ A \rightarrow \mathbb{N}$$

Whereas $\mathit{cost}\ p < \mathit{cost}\ q$ iff p expresses the actual changes *more precisely* than q.

Example

Let p be a patch, imagine we want to transform a *Leaf* into a *Node* x, for some x. There are two patches that could do it:

i) *Del* (*hd* *Leaf*) (*Ins* (*hd* (*Node* x)) p)

ii) *Mod* (*diff* (*hd* *Leaf*) (*hd* (*Node* x))) p

Example

Let p be a patch, imagine we want to transform a *Leaf* into a *Node* x, for some x. There are two patches that could do it:

i) $\mathit{Del}\ (\mathit{hd}\ \mathit{Leaf})\ (\mathit{Ins}\ (\mathit{hd}\ (\mathit{Node}\ x))\ p)$
ii) $\mathit{Mod}\ (\mathit{diff}\ (\mathit{hd}\ \mathit{Leaf})\ (\mathit{hd}\ (\mathit{Node}\ x)))\ p$

We want patch (i) to have lower cost than (ii), as it clearly expresses that the structure of the tree changed! Whereas patch (ii) gives the impression that the contents of a Node changed.

We can calculate a cost function that will select patch (i) instead of patch (ii) and, moreover, makes $\mathit{dist}\ x\ y = \mathit{cost}\ (\mathit{diff}\ x\ y)$ into a metric.

The Generic Diff

Universe of Regular Tree Types:

$$\begin{array}{rcl} T, U & ::= & \top \\ & | & \bot \\ & | & T + U \\ & | & T \times U \\ & | & \mathbb{N} \\ & | & (T\ U) \\ & | & \mu T \end{array}$$

Generic Functions and *Patch*

The definition of *Patch* T follows by induction on T. For example, if $T \equiv U + V$,

$$\begin{aligned} \mathit{Patch}\,(U + V) &\approx (U + V) \times (U + V) \\ &\approx U^2 + 2 \times U \times V + V^2 \\ &\approx \mathit{Patch}\,U + \mathit{Patch}\,V + U \times V + V \times U \end{aligned}$$

Generic Functions and *Patch*

The definition of *Patch* T follows by induction on T. For example, if $T \equiv U + V$,

$$\begin{aligned} \mathit{Patch}\,(U + V) &\approx (U + V) \times (U + V) \\ &\approx U^2 + 2 \times U \times V + V^2 \\ &\approx \mathit{Patch}\ U + \mathit{Patch}\ V + U \times V + V \times U \end{aligned}$$

In order to handle fixpoints generically, we serialize them just like we did with *Tree*s.

Our Contributions (so far)

A short summary of our contributions:

- A notion (and a indexed-datatype) of *Patch* for the universe of RTT.
- Development of a generic *diff* and *apply* for the universe of RTT.
- Definition of a notion of residual, that allows for structural merging. This is what we are currently working on.
- Correctness proofs of our algorithms.

The UNIX `diff`

Looking at the very favorite diffing tool out there, we see that their approach to patches is drastically different!
The typical output from UNIX `diff` will contain a list of:

1. A line number,
2. an edit operation,
3. new content that needs to be inserted.

The UNIX `diff`

Looking at the very favorite diffing tool out there, we see that their approach to patches is drastically different!
The typical output from UNIX `diff` will contain a list of:

1. A line number,
2. an edit operation,
3. new content that needs to be inserted.

The line number is the crucial part!

Since files can be seen as Lists of Lines, the *edit operations* are seen as editing lines and the *location* is the line number!

Patches as Locations and Changes

We would then expect to be able to write our patches in a similar fashion to the UNIX diff:

Patches as Locations and Changes

We would then expect to be able to write our patches in a similar fashion to the UNIX diff:

$$\mathit{Patch}\ T \approx \mathit{List}\ (\exists ty\ .\ ty \leqslant T \times \mathit{Change}\ ty)$$

For some suitable sub-type predicate $\_ \leqslant \_$. Where a proof $p : ty \leqslant T$ would specify a location of ty inside type T.

Patches as Locations and Changes

We would then expect to be able to write our patches in a similar fashion to the UNIX diff:

$$\mathit{Patch}\ T \ \approx\ \mathit{List}\ (\exists ty\ .\ ty\ \leqslant\ T\ \times\ \mathit{Change}\ ty)$$

For some suitable sub-type predicate $\_\ \leqslant\ \_$. Where a proof $p\ :\ ty\ \leqslant\ T$ would specify a location of ty inside type T.

We are left to specify what *Change* should be!

Changes inside a *Tree*

First we fix a $t : \mathit{Tree}\ A$ and study which transformations it can undergo.

Changes inside a *Tree*

First we fix a $t : \mathit{Tree}\ A$ and study which transformations it can undergo.

i) We can always add or remove subtrees from t.

ii) If t is a Node with a value $x : A$ inside, we can modify x and recursively diff the two subtrees of t.

Changes inside a *Tree*

First we fix a $t : \mathit{Tree}\ A$ and study which transformations it can undergo.

i) We can always add or remove subtrees from t.
ii) If t is a *Node* with a value $x : A$ inside, we can modify x and recursively diff the two subtrees of t.

Now it is a matter of writing a type $\mathit{Change}\ (\mathit{Tree}\ A)$ and algorithms that detect and apply these transformations!

Changes inside a *Tree*

Going back to our *Tree* type, a suitable definition of *Change* could be:

```
data Change : Set → Set where
  Atom : A → A → Change A
  Ins  : Ctx (Tree A) → Change (Tree A)
  Del  : Ctx (Tree A) → Change (Tree A)
```

Here *Ctx* stands for one-hole contexts,

Insertions and deletions represent the structural modifications over our *Tree* and *Atom*ic changes are those that change an A inside the tree.

Changes inside a *Tree*

Going back to our *Tree* type, a suitable definition of *Change* could be:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Change} : \mathit{Set} \to \mathit{Set}\ \textbf{where} \\
\quad \mathit{Atom} : A \to A \to \mathit{Change}\ A \\
\quad \mathit{Ins} \;\; : \mathit{Ctx}\,(\mathit{Tree}\ A) \to \mathit{Change}\,(\mathit{Tree}\ A) \\
\quad \mathit{Del} \;\, : \mathit{Ctx}\,(\mathit{Tree}\ A) \to \mathit{Change}\,(\mathit{Tree}\ A)
\end{array}
$$

Here *Ctx* stands for one-hole contexts,

Insertions and deletions represent the structural modifications over our *Tree* and *Atom*ic changes are those that change an A inside the tree.

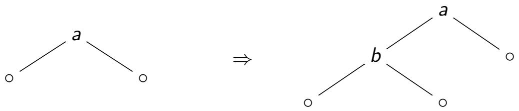
Locations in a tree are trivial: one can go *left* or *right* into a subtree or stay *here* and take the contents of a *Node*.

Problem!

If we try to translate a *PatchTree* to this location-based view we get a problem: it is very hard to extract the contexts of insertions and deletions.

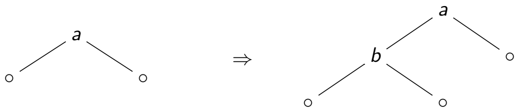
Problem!

If we try to translate a *PatchTree* to this location-based view we get a problem: it is very hard to extract the contexts of insertions and deletions.



Problem!

If we try to translate a *PatchTree* to this location-based view we get a problem: it is very hard to extract the contexts of insertions and deletions.



Three patches can work here:

- $Cpy\ a\ (Ins\ b\ (Ins\ \circ\ (Cpy\ \circ\ (Cpy\ \circ\ Nil))))$
- $Cpy\ a\ (Ins\ b\ (Cpy\ \circ\ (Ins\ \circ\ (Cpy\ \circ\ Nil))))$
- $Cpy\ a\ (Ins\ b\ (Cpy\ \circ\ (Cpy\ \circ\ (Ins\ \circ\ Nil))))$

A Possible Solution

Modify the patch type:

```
data PatchTree (A : Set) : Set where
  Mod : (x y : (TreeF A 1)) (hip : arity x ≡ arity y)
      → Vec (PatchTree A) (arity x)
      → PatchTree A
  Ins : Ctx (Tree A) → PatchTree A → PatchTree A
  Del : Ctx (Tree A) → PatchTree A → PatchTree A
```

A Possible Solution

Modify the patch type:

$$
\begin{aligned}
&\textbf{data}\ \mathit{PatchTree}\ (A : \mathit{Set}) : \mathit{Set}\ \textbf{where}\\
&\quad \mathit{Mod} : (x\ y : (\mathit{TreeF}\ A\ 1))\ (\mathit{hip} : \mathit{arity}\ x \equiv \mathit{arity}\ y)\\
&\qquad\quad \to \mathit{Vec}\ (\mathit{PatchTree}\ A)\ (\mathit{arity}\ x)\\
&\qquad\quad \to \mathit{PatchTree}\ A\\
&\quad \mathit{Ins} : \mathit{Ctx}\ (\mathit{Tree}\ A) \to \mathit{PatchTree}\ A \to \mathit{PatchTree}\ A\\
&\quad \mathit{Del} : \mathit{Ctx}\ (\mathit{Tree}\ A) \to \mathit{PatchTree}\ A \to \mathit{PatchTree}\ A
\end{aligned}
$$

Define *diff* over a *single* *Tree* A and use an oracle to *align* one *Tree* A against a list of *Tree* A.

$$
\begin{aligned}
&\mathit{diff} : \mathit{Tree}\ A \to \mathit{Tree}\ A \to \mathit{PatchTree}\ A\\
&\mathcal{O} : \mathit{Tree}\ A \to \mathit{List}\ (\mathit{Tree}\ A) \to \mathit{Ctx}\ (\mathit{Tree}\ A)
\end{aligned}
$$

The new algorithm

The adapted algorithm follows in pseudo-code:

```
$diff\ x\ y$
    $= choose\ (\{Ins\ (\mathcal{O}\ x\ (ch\ y))\ (diff\ x\ (\mathcal{O}\ x\ (ch\ y) \rhd y))$
                  $\mid if\ arity\ y > 0\}$
      $\cup$       $\{Del\ (\mathcal{O}\ y\ (ch\ x))\ (diff\ (\mathcal{O}\ y\ (ch\ x) \rhd x)\ y)$
                  $\mid if\ arity\ x > 0\}$
      $\cup$       $\{Mod\ (hd\ x)\ (hd\ y)\ (zipWith\ diff\ (ch\ x)\ (ch\ y)$
                  $\mid if\ arity\ x \equiv arity\ y)\}$
```

We denote by $ctx \rhd y$ the operation of matching the *ctx* with y and extracting what is in the *hole*.

Which Algorithm, then?

It is fairly easy to frame this new algorithm in a generic fashion; The functions that manipulate patches become much more complicated, though.

Having an isomorphism between *Patch*es and a list of locations with changes would be great, nevertheless!

Not only type-theory tells us it's all about the structure (which we lose when we serialize the fixpoint in the first approach), but we can borrow many concepts from Separation Logic to reason about *Patch*es.

A Taste of Separation Logic

In the work of Swierstra and Loh [2], we can see how the concept of a separating conjunction is very useful in the context of version control.

Abstractly, given a model M we say

$$\begin{aligned} M \vDash P * Q \quad \text{iff} \quad \exists M_0, M_1. \quad & M = M_0 \mathbin{\dot\cup} M_1 \\ \wedge \quad & M_0 \vDash P \wedge M_1 \vDash Q \end{aligned}$$

Reading it out: A model M satisfies the separating conjunction of P and Q iff it can be written as the disjoint union of two models which satisfy the respective predicates.

A Taste of Separation Logic

If we take the model to be (a suitable representation of) the object under version control and look at the frame rule:

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}\ mod(c) \cap addr(R) = \emptyset$$

where

- $mod(c)$ is the set of locations modified by a patch c
- $addr(R)$ is the set of locations referenced by a formula (depends on the underlying logic).

A Taste of Separation Logic

If we take the model to be (a suitable representation of) the object under version control and look at the frame rule:

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}\ mod(c) \cap addr(R) = \emptyset$$

where

- $mod(c)$ is the set of locations modified by a patch c
- $addr(R)$ is the set of locations referenced by a formula (depends on the underlying logic).

We see we can only borrow from Separation Logic if we have a way of looking at patches as localized changes.

Closing Remarks

- We want to have a notion of location present.
- Speaking about conflicts becomes much simpler!
- Computing a diff becomes more expensive, however (we have less opportunity to memoize calls).
- The problem lies in defining a *Patch* for type application, and, in particular, to least-fixpoints... Regular types are a cake!

Back to CSV files

We began with the following situation:

Name	*Surname*	*Number*	*Grade*
Alice	*Lane*	440	7.5
Bob	*Wright*	593	6.5
Carroll	*Clark*	168	8.5

With CSV files encoded as type:

$$CSV\ A = List\ (List\ A)$$

Back to CSV files

We began with the following situation:

Name	*Surname*	*Number*	*Grade*
Alice	*Lane*	440	7.5
Bob	*Wright*	593	6.5
Carroll	*Clark*	168	8.5

With CSV files encoded as type:

$$\mathit{CSV}\ A \;=\; \mathit{List}\ (\mathit{List}\ A)$$

Prof. Green patch alters the structure: it inserts one element on every inner list;

Prof. Pink edit changes the contents of type *a inside* the inner-most list.

E. Lempsink, S. Leather, and A. 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.

W. Swierstra and A. Löh.
The semantics of version control.
In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '14, pages 43–54, 2014.

L. Bergroth, H. Hakonen, and T. Raita.
A survey of longest common subsequence algorithms.
In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 39–48, 2000.

Structure Aware Version Control

Victor C. Miraldo and Wouter Swierstra

University of Utrecht

13th of October, 2016