The Case Against Edit Scripts

Victor Cacciari Miraldo

December 31, 2019

1 Introduction

Edit scripts are bad. [**(1.1) Victor:**

- Too much redundancy implies expensive algorithms.
- Too restrictive on opeations implies not being able to duplicate or permute.
- When coupled with line-based diff, merges are bad.
- Show a couple examples.

]

We propose an extensional approach.

2 Background

[**(2.1) Victor:** Some edit-scripts; some about tree-diffing]
[**(2.2) Victor:** Primer on unification and substitution and term algebras]

3 Algebra of Extensional Patches

Instead of linearizing trees and relying on very local operations such as insertion, deletions and copying of a single constructor, we can take an extensional look over patches. We can describe patches trough a partial map directly. Take the patch that deletes the left subtree of a binary tree – which can be described by the $Del\ Bin\ (Del\ \ldots\ (Cpy\ \ldots\ Nil))$ edit script. A Haskell function that performs that operation can be given by:

$$\begin{aligned} &delL\ (Bin\ \_\ x) = Just\ x \\ &delL\ \_ = Nothing \end{aligned}$$

The $delL$ function specifies a domain – those trees with a Bin at their root – and a transformation, which forgets the root and its left child, returning only the right child of the root. One way of representing this is by $Bin\ y\ x \mapsto x$,

where x and y are variables, which are bound on the *deletion context* of the patch and used on the *insertion context* of the patch.

Let us look at another example: the patch that swaps the children of a binary tree – which is already impossible to represent with edit-scripts. It could be represented by a Haskell function below, or by $\mathit{Bin}\ x\ y \mapsto \mathit{Bin}\ y\ x$.

$$\mathit{swap}\ (\mathit{Bin}\ x\ y) = \mathit{Just}\ (\mathit{Bin}\ y\ x)$$
$$\mathit{swap}\ \_ = \mathit{Nothing}$$

In the examples above, we have informally described extensional patches over a simple *term algebra*, namelly, that of binary trees with Bin and Leaf. Next we explore this notion of patches for arbitrary term algebras. [**(3.1) Victor:** and our prototype works for mutually recursive bla bla bla]

Because we need the notion of variable to specify our deletion and insertion contexts, we will work with the usual term algebra, but augmented with a countable set V of variables. That is, let L be a language, we denote by $\mathcal{T}_L$ the set of terms over L augmented with the set V of variables. For any $t \in \mathcal{T}_L$, $\mathbf{vars}\ t \subseteq V$ denotes the variables in t. When $\mathbf{vars}\ t = \emptyset$, we say t is a *term*.

Definition 1 (Patch)**.** Let L be a language, a patch p consists in any element of $\mathcal{T}_L \times \mathcal{T}_L$ such that $\mathbf{vars}\ (\mathbf{ins}\ p)\ \subseteq\ (\mathbf{vars}\ (\mathbf{del}\ p))$, where $\mathbf{del}\ p$ and $\mathbf{ins}\ p$ ared $\mathbf{ins}\ p$ and second are the first and second projections, respectivelly. Given two elements d, i of $\mathcal{T}_L$, we denote a patch as $d \mapsto i$.

As usual when working with binders and variables, we assume that variable name clashes between two patches. Application of a patch to a term is easily defined with the help of unification. Take the *swap* patch, $\mathit{Bin}\ x\ y \mapsto \mathit{Bin}\ y\ x$, and the tern $t = \mathit{Bin}\ \mathit{Leaf}\ (\mathit{Bin}\ \mathit{Leaf}\ \mathit{Leaf})$. First we must unify the deletion context wiht t, which yields the substitution $x = \mathit{Leaf} \wedge y = \mathit{Bin}\ \mathit{Leaf}\ \mathit{Leaf}$. To get the result, we must apply this substitution to the insertion context of our patch.

Definition 2 (Application)**.** Let p be a patch over $\mathcal{T}_L$ and t a term over $\mathcal{T}_L$, we say p applies to t whenever $\mathbf{del}\ p$ unifies with t. Let α be such substitution, the result of the application is $\alpha\ (\mathbf{ins}\ p)$. We define the relation $\mathbf{app}\ p\ t\ u$ to captures exactly that.

$$\mathbf{app}\ p\ t\ u \triangleq \exists\ \alpha\ .\ (\mathbf{del}\ p) \cong_\alpha t \wedge \alpha\ (\mathbf{ins}\ p) \equiv u$$

We often abuse notation and write $\mathbf{app}\ p\ t = u$ instead of $\mathbf{app}\ p\ t\ u$.

Lemma 1. *For all patch p, t and u, if* $\mathbf{app}\ p\ t = u$*, then u is a term, that is,* $\mathbf{vars}\ u \equiv \mathit{emptyset}$.

Proof. Let α be the witness of $\mathbf{app}\ p\ t\ u$, we know $u \equiv \alpha\ (\mathbf{ins}\ p)$. We must prove that α substitutes all variables in $\mathbf{ins}\ p$ for terms to conclude the proof. That is simple considering that $(\mathbf{del}\ p) \cong_\alpha t$, $\mathbf{vars}\ t \equiv \mathit{emptyset}$ and $\mathbf{vars}\ (\mathbf{ins}\ p)\ \subseteq\ (\mathbf{vars}\ (\mathbf{del}\ p))$. □

With a notion of application at hand, we can define an extensional equality for our patches. We say patches p and q are equal, denoted $p \sim q$, whenever

$(\textbf{app}\ p\ t\ u) \iff (\textbf{app}\ q\ t\ u)$, for all t, u. It is easy to prove this gives rise to an equivalence relation. Moreover, it correctly identifies patches equal up to renaming of variables.

The next step in constructing an algebra of patches, is to study the composition of patches. [**(3.2) Victor:** hint at optimality problems?] Given patches p and q, however, they are not always composable. Take $p = \mathit{Bin}\ x\ y \mapsto \mathit{Bin}\ y\ x$ and $q = \mathit{Leaf} \mapsto \mathit{Bin}\ \mathit{Leaf}\ \mathit{Leaf}$, $p \circ q$ is defined as $\mathit{Leaf} \mapsto \mathit{Leaf}$, but $q \circ p$ cannot be defined: the result of p has a Bin at its head where q expects a Leaf.

Definition 3 (Composition)**.** Let p and q be patches, we say p composes with q, denoted $\mathit{comp}\ p\ q$, whenever $\textbf{del}\ p$ is unifiable with $\textbf{ins}\ q$. Assume $\mathit{comp}\ p\ q$ and let σ be the substituion witnessing the unification above. We define $p \circ q$ as:

$$p \circ q \triangleq \sigma\ (\textbf{del}\ q) \mapsto \sigma\ (\textbf{ins}\ p)$$

Lemma 2. *Let p and q be patches such that $\mathit{comp}\ p\ q$. Then, for all t, u, $\textbf{app}\ (p \circ q)\ t = u$ if and only if $\textbf{app}\ q\ t = w \wedge \textbf{app}\ p\ w = u$, for some w.*

Proof. todo: transcribe from notebook □

Lemma 3. *For any patch p, the identity patch $x \mapsto x$ is a left and right identity to patch composition, that is, $p \circ (x \mapsto x) \sim p$ and $(x \mapsto x) \circ p \sim p$.*

Proof. trivial □

Before tackling associativity of composition, we must prove two auxiliary lemmas, below.

Lemma 4. *Given p and q composable patches, let $\sigma = \textbf{mgu}(\textbf{del}\ p, \textbf{ins}\ q)$, then there exists σ_p, σ_q such that $\sigma = \sigma_p \cup \sigma_q$ and $\sigma_p\ \textbf{del}\ p = \sigma_q\ \textbf{ins}\ q$.*

Proof. Immediate since $\textbf{vars}\ p \cap \textbf{vars}\ q = \emptyset$. □

Lemma 5. *Given p and q composable patches, let $\sigma = \textbf{mgu}(\textbf{del}\ p, \textbf{ins}\ q)$, then σ is idempodent in $\textbf{del}\ q$ and $\textbf{ins}\ p$. That is, $\sigma\ \sigma\ \textbf{del}\ q = \sigma\ \textbf{del}\ q$ and similarly for $\textbf{ins}\ p$.*

Proof. transcribe □

Finally, we can prove the associativity of our composition operation.

Lemma 6. *Let p and q be composable patches. Let r be a patch composable with $p \circ q$. Then, q and r are composable and p and $q \circ r$ are composable. Moreover, $((p \circ q)) \circ r \sim p \circ ((r \circ q))$*

Proof. transcribe from notebook; somewhat long. □

[**(3.3) Victor:** I still have to talk about inverses, which is just swapping the deletion and insertion cotexts.]

From these results, it follows that patches form a grupoid structure over $\mathcal{T}_L$, for any L.

Yet, from a pratical standpoint, we composition might obfuscate potential shares between the source and destination tree.

4 Merging

[**(4.1) Victor:**

- This construction of patches admits a merge operator.

]

5 Experiments

[**(5.1) Victor:** We are up to 30% success rate on the dataset! yay]

6 Discussion

6.1 The Case Against *cost*

[**(6.1) Victor:**

- Defining the *best* patch is difficult
- The point of the cost, in ES, is to eliminate redundant operations. $cost\ (del\ c\ (ins\ c)) = 2$ whereas $cost\ (cpy\ c) = 0$
- In our case, redundant copies might be present as a byproduct of enforcing the sharing of subtrees.
- From our experimental results, it seems that patches that copy larger subtrees, closer to the root, merge better. Hence, this could better guide a notionof optimality
- We leave this as future work

]