Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Generic diff

Andres Löh

joint work with Eelco Lempsink and Sean Leather

Dept. of Information and Computing Sciences, Utrecht University

IFIP WG 2.1 meeting #64, Weltenburg, April 2, 2009

Overview

- "Normal" diff
- Tree diff
- Generic diff
- Future work

What is diff?

$$\text{diff} : \text{List A} \longrightarrow \text{List A} \longrightarrow \text{Diff A}$$

where A is either Char or String.

What is diff?

> diff : List A → List A → Diff A

where A is either Char or String.

> patch : Diff A → List A → Maybe A

What is diff?

```
diff : List A → List A → Diff A
```

where A is either Char or String.

```
patch : Diff A → List A → Maybe A
```

```
data Diff (A : Set) : Set where
  Mk   : A → Diff → Diff
  Rm   : A → Diff → Diff
  Cp   : A → Diff → Diff      -- or without the A
  Stop : Diff
```

Observations

Minimize size (diff x y), for some definition of size.

$$
\begin{aligned}
&\text{patch (diff x y) x} && \equiv \text{Just y} \\
&\text{patch Stop x} && \equiv \text{Just x} \\
&\text{patch } (d_1 \mathbin{+\!\!+} d_2)\ \text{x} && \equiv \text{patch } d_1\ \text{x} \gg\!= \text{patch } d_2 \\
&\text{patch (reverse (diff y x)) x} && \equiv \text{Just y}
\end{aligned}
$$

Why not normal diff?

Bad description of the change.

Why not normal diff?

Bad description of the change.

Cannot be done in a typed way – to patch:

- serialize,
- patch the string,
- parse (and hope)

Why not normal diff?

Bad description of the change.

Cannot be done in a typed way – to patch:

- serialize,
- patch the string,
- parse (and hope)

Patching may fail, parsing should not.

Running example

```
mutual
  data Expr : Set where
    1    : Expr
    −_   : Expr → Expr
    _+_  : Expr → Expr → Expr
    Let  : Decl → Expr → Expr
  data Decl : Set where
    Val  : Expr → Decl
```

Serves to demonstrate:

- constructors with different numbers of arguments
- mutually recursive types

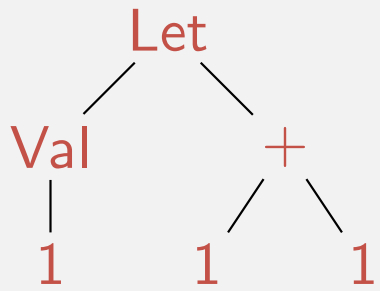
Diff on trees (Lozano, Valiente)

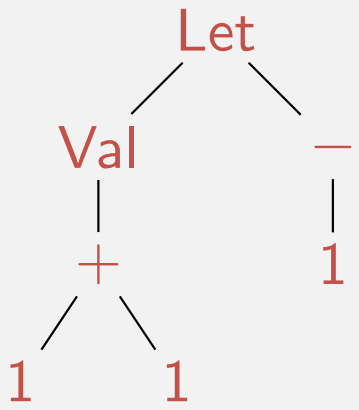
```
data Tree : Set where
  Fork : Label → List Tree → Tree
```

```
diff  : List Tree → List Tree → Diff
patch : Diff → List Tree → Maybe (List Tree)
```

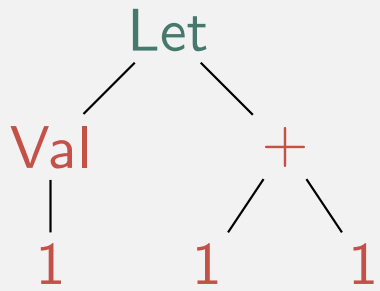
```
data Diff : Set where
  Mk   : Label → Diff → Diff
  Rm   : Label → Diff → Diff
  Cp   : Label → Diff → Diff      -- or without Label
  Stop : Diff
```

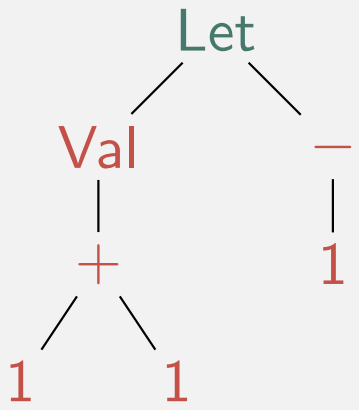
Example



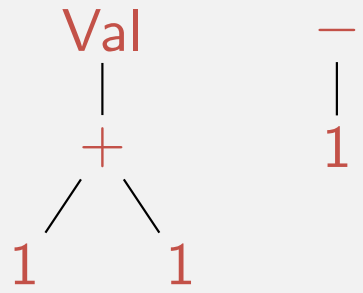


Example



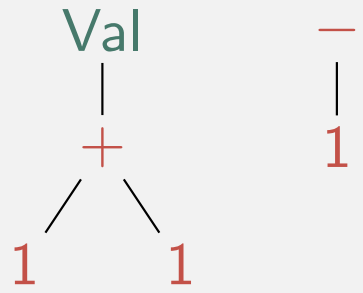


Example



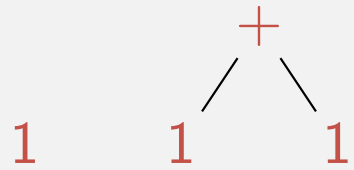
Cp ‘Let’

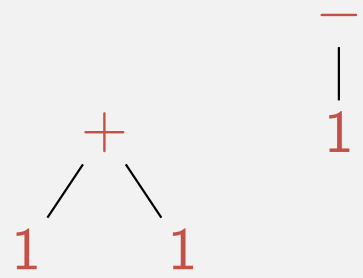
Example



Cp ‘Let’

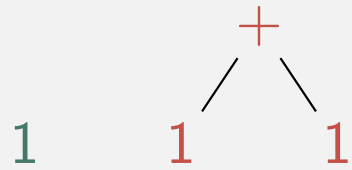
Example

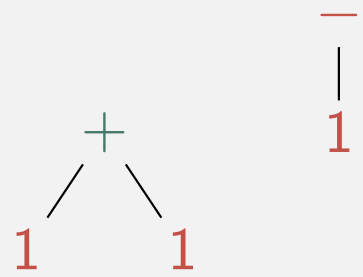




Cp ‘Let’ $ Cp ‘Val’

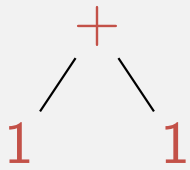
Example

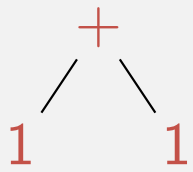




Cp ‘Let’ $ Cp ‘Val’

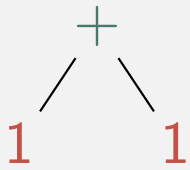
Example

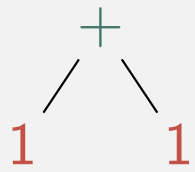


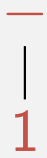


Cp ‘Let’ $ Cp ‘Val’ $ Rm ‘1’

Example

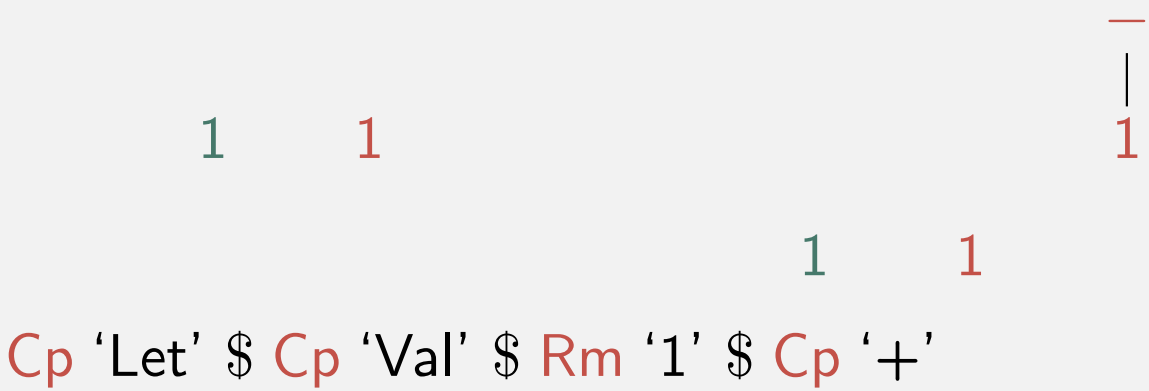






Cp ‘Let’ $ Cp ‘Val’ $ Rm ‘1’

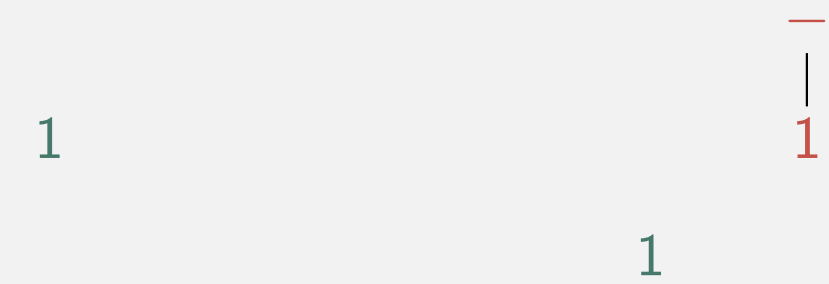
Example



Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Example




```
Cp ‘Let’ $ Cp ‘Val’ $ Rm ‘1’ $ Cp ‘+’
$ Cp ‘1’
```

Example

Cp ‘Let’ $ Cp ‘Val’ $ Rm ‘1’ $ Cp ‘+’
$ Cp ‘1’ $ Cp ‘1’

Example

1

Cp ‘Let’ $ Cp ‘Val’ $ Rm ‘1’ $ Cp ‘+’
$ Cp ‘1’ $ Cp ‘1’ $ Mk ‘−’

Example

Cp ‘Let’ $ Cp ‘Val’ $ Rm ‘1’ $ Cp ‘+’
$ Cp ‘1’ $ Cp ‘1’ $ Mk ‘−’ $ Mk ‘1’

Example

Cp ‘Let’ $ Cp ‘Val’ $ Rm ‘1’ $ Cp ‘+’
$ Cp ‘1’ $ Cp ‘1’ $ Mk ‘−’ $ Mk ‘1’
$ Stop

Observations

Subproblems are given by two forests. Trees in the forest belong to the family of types we consider.

The constructors determine how the forests change.

Both trees (forests) are considered in a depth-first preorder traversal.

The universe

$$
\begin{array}{lcl}
\mathsf{Constr} & : & \mathbb{N} \to \mathsf{Set} \\
\mathsf{Constr\ n} & = & \mathsf{List\ (Fin\ n)} \\
\mathsf{Type} & : & \mathbb{N} \to \mathsf{Set} \\
\mathsf{Type\ n} & = & \mathsf{List\ (Constr\ n)} \\
\mathsf{Fam} & : & \mathbb{N} \to \mathsf{Set} \\
\mathsf{Fam\ n} & = & \mathsf{Vec\ (Type\ n)\ n}
\end{array}
$$

Families are a collection of types.

Types are a collection of constuctors.

Constructors contain fields with recursive calls.

Expressions

Assume that expr is 0 and decl is 1.

```
ExprFam : Fam 2
ExprFam =
          ([]                        ::
           (expr :: [])              ::
           (expr :: expr :: [])      ::
           (decl :: expr :: [])      ::
           []
          ) ::
          ((expr :: [])              ::
           []
          ) ::
          []
```

```
mutual
  data Expr : Set where
    1     : Expr
    −_    : Expr → Expr
    _+_   : Expr → Expr → Expr
    Let   : Decl → Expr → Expr

  data Decl : Set where
    Val   : Expr → Decl
```

Environments

```
data Env {A : Set} (I : A → Set) : List A → Set where
  []     : Env I []
  _::_   : {x : A} {xs : List A} →
           I x → Env I xs → Env I (x :: xs)
```

Lists indexed by the list of types of their elements.

Interpreting the universe

$$
\begin{aligned}
&\mathsf{C}[\![\_]\!] : \{\mathsf{n} : \mathbb{N}\} \to \\
&\qquad \mathsf{Constr}\ \mathsf{n} \to (\mathsf{Fin}\ \mathsf{n} \to \mathsf{Set}) \to \mathsf{Set} \\
&\mathsf{C}[\![\_]\!]\ \{\mathsf{n}\}\ \mathsf{xs}\ \mathsf{f} = \mathsf{Env}\ \mathsf{f}\ \mathsf{xs}
\end{aligned}
$$

Interpreting the universe

```
C⟦_⟧ : {n : ℕ} →
       Constr n → (Fin n → Set) → Set
C⟦_⟧ {n} xs f = Env f xs

T⟦_⟧ : {n : ℕ} →
       Type n → (Fin n → Set) → Set
T⟦_⟧ {n} xs f = Σ (Fin (length xs))
                  (λ n → C⟦ lookup n (fromList xs) ⟧ f)
```

Interpreting the universe

$$
\begin{aligned}
&\mathsf{C}[\![\_]\!] : \{\mathsf{n} : \mathbb{N}\} \to \\
&\qquad \mathsf{Constr\ n} \to (\mathsf{Fin\ n} \to \mathsf{Set}) \to \mathsf{Set} \\
&\mathsf{C}[\![\_]\!]\ \{\mathsf{n}\}\ \mathsf{xs\ f} = \mathsf{Env\ f\ xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{T}[\![\_]\!] : \{\mathsf{n} : \mathbb{N}\} \to \\
&\qquad \mathsf{Type\ n} \to (\mathsf{Fin\ n} \to \mathsf{Set}) \to \mathsf{Set} \\
&\mathsf{T}[\![\_]\!]\ \{\mathsf{n}\}\ \mathsf{xs\ f} = \Sigma\ (\mathsf{Fin\ (length\ xs)}) \\
&\qquad\qquad\qquad\qquad (\lambda\ \mathsf{n} \to \mathsf{C}[\![\ \mathsf{lookup\ n\ (fromList\ xs)}\ ]\!]\ \mathsf{f})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{F}[\![\_]\!] : \{\mathsf{n} : \mathbb{N}\} \to \\
&\qquad \mathsf{Fam\ n} \to (\mathsf{Fin\ n} \to \mathsf{Set}) \to \mathsf{Fin\ n} \to \mathsf{Set} \\
&\mathsf{F}[\![\_]\!]\ \mathsf{xs\ f\ fn} = \mathsf{T}[\![\ \mathsf{lookup\ fn\ xs}\ ]\!]\ \mathsf{f}
\end{aligned}
$$

Interpreting the universe

```
C⟦_⟧ : {n : ℕ} →
       Constr n → (Fin n → Set) → Set
C⟦_⟧ {n} xs f = Env f xs

T⟦_⟧ : {n : ℕ} →
       Type n → (Fin n → Set) → Set
T⟦_⟧ {n} xs f = Σ (Fin (length xs))
                  (λ n → C⟦ lookup n (fromList xs) ⟧ f)

F⟦_⟧ : {n : ℕ} →
       Fam n → (Fin n → Set) → Fin n → Set
F⟦_⟧ xs f fn  = T⟦ lookup fn xs ⟧ f

data μ {n : ℕ} (F : Fam n) (fn : Fin n) : Set where
  ⟨_⟩ : F⟦ F ⟧ (μ F) fn → μ F fn
```

Fixing a specific family

```
module GenericDiff {n : ℕ} (F : Fam n) where
...
```

Helpers

The type of constructors of a type.

```
constrOf : Fin n → Set
constrOf t = Fin (length (lookup t F))
```

Given a type and a constructor of that type, the fields.

```
fields : (t : Fin n) → constrOf t → List (Fin n)
fields t c = lookup c (fromList (lookup t F))
```

Generic diff

```
data Diff : List (Fin n) → List (Fin n) → Set where
  Mk   : {xs ys : List (Fin n)} →
         (y : Fin n) → (c : constrOf y) →
         Diff xs (fields y c ++ ys) →
         Diff xs (y :: ys)
  Rm   : {xs ys : List (Fin n)} →
         (x : Fin n) → (c : constrOf x) →
         Diff (fields x c ++ xs) ys →
         Diff (x :: xs)          ys
  Cp   : {xs ys : List (Fin n)} →
         (z : Fin n) → (c : constrOf z) →
         Diff (fields z c ++ xs) (fields z c ++ ys) →
         Diff (z :: xs)          (z :: ys)
  Stop : Diff []                 []
```

Generic diff

$$
\begin{aligned}
&\mu\mathsf{Env} : \quad \mathsf{List}\ (\mathsf{Fin}\ \mathsf{n}) \rightarrow \mathsf{Set} \\
&\mu\mathsf{Env} = \mathsf{Env}\ (\mu\ \mathsf{F}) \\
&\\
&\mathsf{patch} : \quad \{\mathsf{xs}\ \mathsf{ys} : \mathsf{List}\ (\mathsf{Fin}\ \mathsf{n})\} \rightarrow \\
&\qquad\qquad \mathsf{Diff}\ \mathsf{xs}\ \mathsf{ys} \rightarrow \mu\mathsf{Env}\ \mathsf{xs} \rightarrow \mathsf{Maybe}\ (\mu\mathsf{Env}\ \mathsf{ys}) \\
&\mathsf{diff} : \quad \{\mathsf{xs}\ \mathsf{ys} : \mathsf{List}\ (\mathsf{Fin}\ \mathsf{n})\} \rightarrow \\
&\qquad\qquad \mu\mathsf{Env}\ \mathsf{xs} \rightarrow \mu\mathsf{Env}\ \mathsf{ys} \rightarrow \mathsf{Diff}\ \mathsf{xs}\ \mathsf{ys}
\end{aligned}
$$

How to define patch

```
patch : {xs ys : List (Fin n)} →
        Diff xs ys → μEnv xs → Maybe (μEnv ys)

patch (Rm t c ds) (⟨ c', args ⟩ :: ts) with c' ≟ c
patch (Rm t c ds) (⟨ c', args ⟩ :: ts) | No _      = Nothing
patch (Rm t c ds) (⟨ .c, args ⟩ :: ts) | Yes Refl = patch ds (args ++ ts)

patch (Mk t c ds) ts with patch ds ts
...                  | Nothing = Nothing
...                  | Just ts' with splitEnv (fields t c) ts'
patch (Mk t c ds) ts | Just ._ | (args ++′ rest) = Just (⟨ c, args ⟩ :: rest)
```

What more?

- Constants (then we can recover the "normal" diff)
- Compression (copy entire subtrees)
- Efficiency (memoization)
- Haskell
- Heuristics
- Other notions of Diff