Generic Representations of Tree Transformations

José Pedro Magalhães
Joint work with Jeroen Bransen

Department of Computer Science
University of Oxford

September 28, 2013
What are transformations?

Consider an example datatype:

```
data Expr = Var String
          | Const Int
          | Neg Expr
          | Add Expr Expr
```
What are transformations?

Consider an example datatype:

```
data Expr = Var String
           | Const Int
           | Neg Expr
           | Add Expr Expr
```

Here are some transformations:

- \( \text{Var "a"} \rightarrow \text{Neg (Var "a")} \)
What are transformations?

Consider an example datatype:

```
data Expr = Var String
          | Const Int
          | Neg Expr
          | Add Expr Expr
```

Here are some transformations:

- \(\text{Var "a"} \rightarrow \text{Neg (Var "a")}\)
- \(\text{Add a b} \rightarrow \text{Add b a}\)
What are transformations?

Consider an example datatype:

```
data Expr = Var String |
            Const Int |
            Neg Expr |
            Add Expr Expr
```

Here are some transformations:

- \( \text{Var} \ "a" \mapsto \text{Neg} (\text{Var} \ "a") \)
- \( \text{Add} \ a \ b \mapsto \text{Add} \ b \ a \)
- \( \text{Add} \ a (\text{Add} \ b \ c) \mapsto \text{Add} (\text{Add} \ a \ b) \ c \)
What are transformations?

Consider an example datatype:

```
data Expr = Var String
         | Const Int
         | Neg Expr
         | Add Expr Expr
```

Here are some transformations:

- \( \text{Var "a" } \mapsto \text{Neg (Var "a")} \)
- \( \text{Add a b } \mapsto \text{Add b a} \)
- \( \text{Add a (Add b c) } \mapsto \text{Add (Add a b) c} \)
- \( \text{Add a (Const 0) } \mapsto \text{Add a a} \)
What are transformations?

Consider an example datatype:

\[
data \text{Expr} = \text{Var} \text{ String} \\
| \text{Const} \text{ Int} \\
| \text{Neg} \text{ Expr} \\
| \text{Add} \text{ Expr Expr}
\]

Here are some transformations:

- \( \text{Var } "a" \xrightarrow{} \text{Neg } (\text{Var } "a") \)
- \( \text{Add } a \ b \xrightarrow{} \text{Add } b \ a \)
- \( \text{Add } a \ (\text{Add } b \ c) \xrightarrow{} \text{Add } (\text{Add } a \ b) \ c \)
- \( \text{Add } a \ (\text{Const } 0) \xrightarrow{} \text{Add } a \ a \)
- \( \text{Neg } \text{Neg } \text{Neg } \text{Neg } \text{Neg } \text{Neg } (\text{Const } 1)))))\)
  \xrightarrow{}
  \text{Neg } \text{Neg } \text{Neg } \text{Neg } \text{Neg } \text{Neg } (\text{Const } 2)))))\)
Why do we need to represent transformations?

Large applications need to *transform, edit, or evolve* data:

**Structure editors** A structure editor is a type of editor that is aware of the underlying structure of the document being edited.
Why do we need to represent transformations?

Large applications need to *transform, edit, or evolve* data:

**Structure editors** A structure editor is a type of editor that is aware of the underlying structure of the document being edited.

**Exercise assistants** An exercise assistant is a tool to help students understand and apply fundamental concepts in a given domain.
Why do we need to represent transformations?

Large applications need to transform, edit, or evolve data:

**Structure editors** A structure editor is a type of editor that is aware of the underlying structure of the document being edited.

**Exercise assistants** An exercise assistant is a tool to help students understand and apply fundamental concepts in a given domain.

**Incremental computations** To avoid expensive recomputation in unchanged parts of data, an incremental computation keeps track of changes.
Consider the edit script resulting from computing the difference between \( \text{Add} (\text{Var} \ "a") (\text{Var} \ "b") \) and \( \text{Add} (\text{Var} \ "b") (\text{Var} \ "a") \):

\[
\text{Cpy Add} \$ \text{Cpy Var} \$ \text{Ins} "b" \$ \text{Ins Var} \$
\text{Cpy} "a" \$ \text{Del Var} \$ \text{Del} "b" \$ \text{End}
\]

This edit script does not keep track of the fact that the inserted expressions are not “new”, losing adequate sharing between transformations.
What we need

A representation of transformations that:
- Is as abstract as possible as to what type of transformations are allowed
- Keeps track of subexpression sharing
- Minimises duplication of information
Three approaches

We present three different approaches to the problem, of varying complexity, flexibility, and user-friendliness:

1. Zipper with state
2. Rewriting
3. Explicit encoding

Different approaches might be better suited for one particular application area or another.
Obviously, our approach is datatype-generic. For presentation purposes we use the regular library:

```haskell
instance Regular Expr where
  from (Var s) = L (K s)
  from (Const i) = R (L (K i))
  from (Neg e) = R (R (L (I e)))
  from (Add e1 e2) = R (R (R (L (I e1 :×: I e2))))
  to (L (K s)) = Var s
  to (R (L (K i))) = Const i
  to (R (L (I e)))) = Neg e
  to (R (R (R (L (I e1 :×: I e2)))))) = Add e1 e2
```
1. Zipper with state
1. Zipper with state

Our first approach combines a zipper with state in a monad:

\[ \text{insert} :: \text{Maybe} \ \text{Expr} \]
\[ \text{insert} = \text{navigateZS} (\text{Add} (\text{Const} 1) (\text{Var} "a")) \]$ \[ \text{do} \ \text{downZS} \]
\[ \ \text{rightZS} \]
\[ \ \text{updateZS} \text{ Neg} \]

\[ \text{insert} \equiv \text{Just} (\text{Add} (\text{Const} 1) (\text{Neg} (\text{Var} "a"))) \]
1. Zipper with state

Our first approach combines a zipper with state in a monad:

\[
\text{insert} :: \text{Maybe Expr} \\
\text{insert} = \text{navigateZS} (\text{Add (Const 1) (Var "a"))} \$ \\
\text{do downZS} \\
\text{rightZS} \\
\text{updateZS Neg}
\]

\[
\text{insert} \equiv \text{Just (Add (Const 1) (Neg (Var "a")))}
\]

\[
\text{delete} :: \text{Maybe Expr} \\
\text{delete} = \text{navigateZS (Add (Const 1) (Neg (Var "a")))} \$ \\
\text{do downZS} \\
\text{rightZS} \\
\text{r} \leftarrow \text{downZS} \\
\text{upZS} \\
\text{updateZS (const r)}
\]

\[
\text{delete} \equiv \text{Just (Add (Const 1) (Var "a"))}
\]
1. A zipper

The zipper we use is entirely standard:

```haskell
import Control.Monad

data Loc α where
  Loc :: (Regular α, Zipper (PF α))
  ⇒ α → [Ctx (PF α) α] → Loc α

data family Ctx (φ :: ⋆ → ⋆) :: ⋆ → ⋆

enter :: (Regular α, Zipper (PF α)) ⇒ α → Loc α
leave :: Loc α → α
up, down, left, right :: Loc α → Maybe (Loc α)
on :: Loc α → α
updateM :: Monad φ ⇒ (α → φ α) → Loc α → φ (Loc α)
```
1. Adding state to a zipper

Now we embed this zipper in a state monad:

```plaintext
type ZipperMonad α β = StateT (Loc α) Maybe β
```

Note that \( \text{ZipperMonad } α β \approx \text{Loc } α \rightarrow \text{Maybe } (β, \text{Loc } α) \).
1. Adding state to a zipper

Now we embed this zipper in a state monad:

```haskell
type ZipperMonad α β = StateT (Loc α) Maybe β
```

Note that $\text{ZipperMonad } α β \approx \text{Loc } α \rightarrow \text{Maybe } (β, \text{Loc } α)$.

Moving around:

```haskell
moveZS :: (Loc α \rightarrow \text{Maybe} (Loc α)) \rightarrow \text{ZipperMonad } α α
moveZS m = StateT \$ \lambda l \rightarrow m l >\rightarrow \lambda l' \rightarrow \text{return} (\text{on } l', l')

upZS, downZS, leftZS, rightZS :: \text{ZipperMonad } α α
upZS = moveZS up
downZS = moveZS down
leftZS = moveZS left
rightZS = moveZS right
```
1. Adding state to a zipper II

Updating the value at the current location:

\[
updateZS :: (\alpha \to \alpha) \to \text{ZipperMonad} \; \alpha \; \alpha
\]

\[
updateZS \; f = \text{do} \; l \leftarrow \text{get} \\
\text{let } \text{Just} \; l' = updateM (\text{Just} \circ f) \; l \\
\text{put} \; l' \\
\text{return} \; (\text{on} \; l')
\]
1. Another example

And that’s all that’s needed!

Another example, this time a swap:

\[
\begin{align*}
\text{swap} :: & \text{Maybe Expr} \\
\text{swap} &= \text{navigateZS} (\text{Add} (\text{Const} 1) (\text{Var} "a"))$ \\
\text{do} & \ l \leftarrow \text{downZS} \\
& \ r \leftarrow \text{rightZS} \\
& \ \text{updateZS} (\text{const} \ l) \\
& \ \text{leftZS} \\
& \ \text{updateZS} (\text{const} \ r)
\end{align*}
\]

\[
\text{swap} \equiv \text{Just} (\text{Add} (\text{Var} "a") (\text{Const} 1))
\]
1. Summary

A summary of the representation of transformations using a zipper with state:

- Easy to use
- Simple encoding
- Suitable for structure editors
- Transformations can be verbose
- Transformations cannot be inspected
Our second approach uses a zipper to focus on a specific location, and rewrite rules to apply transformations:

\[
\begin{align*}
\text{insert} & :: \text{Maybe Expr} \\
\text{insert} & = \text{apply} \ [(\text{down} \Rightarrow \text{right}, \text{addNeg})] \ ((\text{Add (Const 1) (Var "a"})) \\
& \text{where addNeg} :: \text{Rule Expr} \\
& \text{addNeg} = \text{rule} \ \lambda x \to x \Rightarrow: \text{Neg} \ x
\end{align*}
\]

\[
\text{insert} \equiv \text{Just (Add (Const 1) (Neg (Var "a")))}
\]

This is a combination of a zipper with generic rewrite rules (Van Noort et al., WGP 2008).
2. Meta-variable extension

Rewrite rules rely on meta-variable extension:

\[
\begin{align*}
\text{type} & \quad Ext \; \phi = K \; Metavar :+: \phi \\
\text{type} & \quad Metavar = Int \\
\text{data} & \quad \mu \phi = \text{In} (\phi (\mu \phi)) \\
\text{type} & \quad Scheme \; \phi = \mu (Ext \; \phi) \\
\text{type} & \quad SchemeOf \; \alpha = Scheme (PF \; \alpha)
\end{align*}
\]
2. More examples

Deleting a node:

\[
\begin{align*}
\text{delete} &:: \text{Maybe Expr} \\
\text{delete} &\equiv \text{apply}\ [(\downarrow \Rightarrow \text{right}, \text{removeNeg})]\ \text{expr2} \text{ where} \\
\text{removeNeg} &:: \text{Rule Expr} \\
\text{removeNeg} &\equiv \text{rule} \lambda x \to \text{Neg x} :\rightsquigarrow : x
\end{align*}
\]
2. More examples

Deleting a node:

\[
del :: \text{Maybe Expr} \\
del = \text{apply \{(down} \Rightarrow \text{right}, \text{removeNeg}\)} \text{ expr2 where} \\
\text{removeNeg :: Rule Expr} \\
\text{removeNeg = rule } \lambda x \rightarrow \text{Neg } x : \Rightarrow : x
\]

And here is the swap example:

\[
\text{swapExpr :: Rule Expr} \\
\text{swapExpr = rule } \lambda a b \rightarrow \text{Add } a b : \Rightarrow : \text{Add } b a
\]
2. Summary

A summary of the representation of transformations using rewrite rules:

- Very easy to use
- Very simple encoding
- Suitable for exercise assistants
- Not all sharing is captured
- The representation of transformations is inconvenient:

\[
\text{expandedSwap} :: \text{Rule Expr} \\
\text{expandedSwap} = \\
\text{In } (R (R (R (R}
\text{ (I (In (L (K 0)))) :×: I (In (L (K 1)))))))
\text{ :→: In } (R (R (R (R}
\text{ (I (In (L (K 1)))) :×: I (In (L (K 0)))))))
\]
3. Explicit encoding

Our third approach represents transformations using explicit paths to shared subterms:

\[
\text{insert} :: \text{Maybe Expr} \\
\text{insert} = \text{apply} \ \text{addNeg} (\text{Add} (\text{Const} \ 1) (\text{Var} \ "a")) \ \text{where} \\
\text{addNeg} :: \text{Transformation Expr} \\
\text{addNeg} = [[[1], \text{Neg} (\text{Ref} [1])]]
\]
3. Explicit encoding

Our third approach represents transformations using explicit paths to shared subterms:

\[\text{insert} :: \text{Maybe Expr}\]
\[\text{insert} = \text{apply } \text{addNeg} (\text{Add } (\text{Const } 1) (\text{Var } "a")) \text{ where}\]
\[\text{addNeg} :: \text{Transformation Expr}\]
\[\text{addNeg} = \begin{bmatrix}1, \text{Neg} (\text{Ref } [1])\end{bmatrix}\]

\[\text{delete} :: \text{Maybe Expr}\]
\[\text{delete} = \text{apply } \text{delNeg} (\text{Add } (\text{Const } 1) (\text{Neg} (\text{Var } "a"))) \text{ where}\]
\[\text{delNeg} :: \text{Transformation Expr}\]
\[\text{delNeg} = \begin{bmatrix}1, \text{Ref } [1,0]\end{bmatrix}\]
3. Explicit encoding

Our third approach represents transformations using explicit paths to shared subterms:

\[
\begin{align*}
\text{insert} :: & \text{Maybe Expr} \\
\text{insert} = & \text{apply } \text{addNeg} (\text{Add (Const 1) (Var "a")}) \text{ where} \\
\text{addNeg} :: & \text{Transformation Expr} \\
\text{addNeg} = & \left[ ([1], \text{Neg (Ref [1])}) \right] \\
\end{align*}
\]

\[
\begin{align*}
\text{delete} :: & \text{Maybe Expr} \\
\text{delete} = & \text{apply } \text{delNeg} (\text{Add (Const 1) (Neg (Var "a")}) \text{ where} \\
\text{delNeg} :: & \text{Transformation Expr} \\
\text{delNeg} = & \left[ ([1], \text{Ref [1,0]}) \right] \\
\end{align*}
\]

\[
\begin{align*}
\text{swap} :: & \text{Maybe Expr} \\
\text{swap} = & \text{apply } \text{swap'} (\text{Add (Const 1) (Var "a")}) \text{ where} \\
\text{swap'} :: & \text{Transformation Expr} \\
\text{swap'} = & \left[ ([0], \text{Ref [1]}), ([1], \text{Ref [0]}) \right] \\
\end{align*}
\]
3. Paths, references, and transformations

Paths are encoded as lists of integers:

\[
\text{type } Path = [\text{Int}]
\]

Yes, this is suboptimal.
3. Paths, references, and transformations

Paths are encoded as lists of integers:

\[
\text{type } \text{Path} = [\text{Int}]
\]

Yes, this is suboptimal.

References are encoded similarly to meta-variables for rewriting:

\[
\text{data } \text{WithRef } \alpha \beta = \text{InR} (\text{PF } \alpha \beta) \\
| \quad \text{Ref } \text{Path}
\]
3. Paths, references, and transformations

Paths are encoded as lists of integers:

\[
\text{type } Path = [\text{Int}]
\]

Yes, this is suboptimal.

References are encoded similarly to meta-variables for rewriting:

\[
data WithRef \; \alpha \; \beta = \text{InR} \; (PF \; \alpha \; \beta) \\
| \; \text{Ref} \; Path
\]

Transformations are then lists of pairs containing paths and values extended with references:

\[
\text{type } Transformation \; \alpha = [(Path, \mu (WithRef \; \alpha))]
\]
3. Ease of use

We get the same “ease of use” problem as with rewriting:

```haskell
let expr_1 = Add (Const 1) (Var "a")

insert :: Maybe Expr
insert = apply addNeg expr_1 where
addNeg :: Transformation Expr
addNeg = [(1, Neg $ Ref [1])]
```
3. Ease of use

We get the same “ease of use” problem as with rewriting:

\[
\text{let } expr_1 = \text{Add (Const 1) (Var "a")}
\]

\[
\text{insert :: Maybe Expr}
\]
\[
\text{insert} = \text{apply addNeg expr_1 where}
\]
\[
\text{addNeg :: Transformation Expr}
\]
\[
\text{addNeg} = \left[[1], \text{In} \circ \text{InR} \circ R \circ L \circ I \circ \text{In} \, \text{Ref}\right]
\]

We solve it by adding a layer of Template Haskell for convenience:

\[
\text{insert :: Maybe Expr}
\]
\[
\text{insert} = \text{apply (fromTransformation\textsubscript{EH} addNeg)} \, expr_1 \text{ where}
\]
\[
\text{addNeg :: Transformation\textsubscript{EH} Expr}
\]
\[
\text{addNeg} = \left[[1], \text{Neg\textsubscript{EH}} (\text{Ref}\textsubscript{EH} [1])\right]
\]
Unlike the previous approaches, this time we have a
\[ \text{diff} :: ( \text{Regular } \alpha, \ldots ) \Rightarrow \alpha \rightarrow \alpha \rightarrow \text{Transformation } \alpha \] such that:

\[ \forall a, b. \ \text{apply} \ (\text{diff } a \ b) \ a \equiv \text{Just } b \]

Our \textit{diff} returns \textit{Transformations} with maximal use of sharing (but other algorithms are possible). More details in the paper.
3. Summary

A summary of the explicit representation of transformations:

👍 Sharing is explicit
👍 We can provide *diff*
👍 Transformations are easy to inspect
👎 Interface is only convenient through Template Haskell
Conclusion

Summarising:

▶ Representing transformations effectively is important
▶ We show three different ways of doing it
▶ All generic, working on families of datatypes (using multirec)
▶ A nice combination of multiple generic programming techniques

All the code is available at http://hackage.haskell.org/package/transformations.
Summarising:

- Representing transformations effectively is important
- We show three different ways of doing it
- All generic, working on families of datatypes (using `multirec`)
- A nice combination of multiple generic programming techniques

All the code is available at
Conclusion

Summarising:

- Representing transformations effectively is important
- We show three different ways of doing it
- All generic, working on families of datatypes (using `multirec`)
- A nice combination of multiple generic programming techniques

All the code is available at

Questions?