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:

$$
\begin{aligned}
\textbf{data}\ \mathit{Expr} = {} & \mathit{Var}\ \ \ \mathit{String} \\
\mid {} & \mathit{Const}\ \mathit{Int} \\
\mid {} & \mathit{Neg}\ \ \ \mathit{Expr} \\
\mid {} & \mathit{Add}\ \ \ \mathit{Expr}\ \mathit{Expr}
\end{aligned}
$$

What are transformations?

Consider an example datatype:

$$\begin{aligned}\textbf{data}\ \mathit{Expr} = {} & \mathit{Var}\ \ \mathit{String} \\ \mid {} & \mathit{Const}\ \mathit{Int} \\ \mid {} & \mathit{Neg}\ \ \mathit{Expr} \\ \mid {} & \mathit{Add}\ \ \mathit{Expr}\ \mathit{Expr}\end{aligned}$$

Here are some transformations:

- $\mathit{Var}\ \texttt{"a"} \leadsto \mathit{Neg}\ (\mathit{Var}\ \texttt{"a"})$

What are transformations?

Consider an example datatype:

$$\begin{aligned}\textbf{data}\ \mathit{Expr} = {} & \mathit{Var}\ \ \mathit{String} \\ \mid {} & \mathit{Const}\ \mathit{Int} \\ \mid {} & \mathit{Neg}\ \ \mathit{Expr} \\ \mid {} & \mathit{Add}\ \ \mathit{Expr}\ \mathit{Expr}\end{aligned}$$

Here are some transformations:

- $\mathit{Var}\ \texttt{"a"} \rightsquigarrow \mathit{Neg}\ (\mathit{Var}\ \texttt{"a"})$
- $\mathit{Add}\ a\ b \rightsquigarrow \mathit{Add}\ b\ a$

What are transformations?

Consider an example datatype:

$$\begin{aligned}\textbf{data}\ \mathit{Expr} = {} & \mathit{Var}\ \ \mathit{String} \\ \mid {} & \mathit{Const}\ \mathit{Int} \\ \mid {} & \mathit{Neg}\ \ \mathit{Expr} \\ \mid {} & \mathit{Add}\ \ \mathit{Expr}\ \mathit{Expr}\end{aligned}$$

Here are some transformations:

- $\mathit{Var}\ \texttt{"a"} \leadsto \mathit{Neg}\ (\mathit{Var}\ \texttt{"a"})$
- $\mathit{Add}\ a\ b \leadsto \mathit{Add}\ b\ a$
- $\mathit{Add}\ a\ (\mathit{Add}\ b\ c) \leadsto \mathit{Add}\ (\mathit{Add}\ a\ b)\ c$

What are transformations?

Consider an example datatype:

$$
\begin{aligned}
\textbf{data}\ \mathit{Expr} = {} & \mathit{Var}\quad \mathit{String} \\
\mid {} & \mathit{Const}\ \mathit{Int} \\
\mid {} & \mathit{Neg}\quad \mathit{Expr} \\
\mid {} & \mathit{Add}\quad \mathit{Expr}\ \mathit{Expr}
\end{aligned}
$$

Here are some transformations:

- $\mathit{Var}\ \texttt{"a"} \rightsquigarrow \mathit{Neg}\ (\mathit{Var}\ \texttt{"a"})$
- $\mathit{Add}\ a\ b \rightsquigarrow \mathit{Add}\ b\ a$
- $\mathit{Add}\ a\ (\mathit{Add}\ b\ c) \rightsquigarrow \mathit{Add}\ (\mathit{Add}\ a\ b)\ c$
- $\mathit{Add}\ a\ (\mathit{Const}\ 0) \rightsquigarrow \mathit{Add}\ a\ a$

What are transformations?

Consider an example datatype:

$$\begin{aligned}\textbf{data}\ \mathit{Expr} = &\ \mathit{Var}\quad \mathit{String}\\ |&\ \mathit{Const}\ \mathit{Int}\\ |&\ \mathit{Neg}\quad \mathit{Expr}\\ |&\ \mathit{Add}\quad \mathit{Expr}\ \mathit{Expr}\end{aligned}$$

Here are some transformations:

- Var "a" $\rightsquigarrow \mathit{Neg}\ (\mathit{Var}$ "a"$)$
- $\mathit{Add}\ a\ b \rightsquigarrow \mathit{Add}\ b\ a$
- $\mathit{Add}\ a\ (\mathit{Add}\ b\ c) \rightsquigarrow \mathit{Add}\ (\mathit{Add}\ a\ b)\ c$
- $\mathit{Add}\ a\ (\mathit{Const}\ 0) \rightsquigarrow \mathit{Add}\ a\ a$
- $\quad \mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Const}\ 1)))))))$
  $\rightsquigarrow \mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Neg}\ (\mathit{Const}\ 2)))))))$

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.

Diff is not enough

Consider the edit script resulting from computing the difference between *Add* (*Var* `"a"`) (*Var* `"b"`) and *Add* (*Var* `"b"`) (*Var* `"a"`):

Cpy Add \$ *Cpy Var* \$ *Ins* `"b"` \$ *Ins Var* \$
Cpy `"a"` \$ *Del Var* \$ *Del* `"b"` \$ *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.

Underlying GP library: `regular`

Obviously, our approach is datatype-generic. For presentation purposes we use the `regular` library:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Regular}\ \mathit{Expr}\ \textbf{where} \\
\quad \textbf{type}\ \mathit{PF}\ \mathit{Expr} = K\ \mathit{String}\ {:}{+}{:}\ K\ \mathit{Int}\ {:}{+}{:}\ I\ {:}{+}{:}\ (I\ {:}{\times}{:}\ I) \\
\quad \mathit{from}\ (\mathit{Var}\ s) \qquad\quad = L\ (K\ s) \\
\quad \mathit{from}\ (\mathit{Const}\ i) \quad\ \ = R\ (L\ (K\ i)) \\
\quad \mathit{from}\ (\mathit{Neg}\ e) \qquad\ = R\ (R\ (L\ (I\ e))) \\
\quad \mathit{from}\ (\mathit{Add}\ e_1\ e_2) = R\ (R\ (R\ (I\ e_1\ {:}{\times}{:}\ I\ e_2))) \\
\quad \mathit{to}\ (L\ (K\ s)) \qquad\qquad\qquad\quad\ = \mathit{Var}\ s \\
\quad \mathit{to}\ (R\ (L\ (K\ i))) \qquad\qquad\quad\ \ = \mathit{Const}\ i \\
\quad \mathit{to}\ (R\ (R\ (L\ (I\ e)))) \qquad\quad\ \ = \mathit{Neg}\ e \\
\quad \mathit{to}\ (R\ (R\ (R\ (I\ e_1\ {:}{\times}{:}\ I\ e_2)))) = \mathit{Add}\ e_1\ e_2
\end{array}
$$

1. Zipper with state

1. Zipper with state

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

```
insert :: Maybe Expr
insert = navigateZS (Add (Const 1) (Var "a")) $
  do downZS
     rightZS
     updateZS Neg
```

$$insert \equiv Just\ (Add\ (Const\ 1)\ (Neg\ (Var\ \texttt{"a"})))$$

1. Zipper with state

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

```
insert :: Maybe Expr
insert = navigateZS (Add (Const 1) (Var "a")) $
  do downZS
     rightZS
     updateZS Neg

insert ≡ Just (Add (Const 1) (Neg (Var "a")))

delete :: Maybe Expr
delete = navigateZS (Add (Const 1) (Neg (Var "a")) $
  do downZS
     rightZS
     r ← downZS
     upZS
     updateZS (const r)

delete ≡ Just (Add (Const 1) (Var "a"))
```

1. A zipper

The zipper we use is entirely standard:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Loc}\ \alpha\ \textbf{where} \\
&\quad \mathit{Loc} :: (\mathit{Regular}\ \alpha, \mathit{Zipper}\ (\mathit{PF}\ \alpha)) \\
&\qquad\quad \Rightarrow \alpha \rightarrow [\,\mathit{Ctx}\ (\mathit{PF}\ \alpha)\ \alpha\,] \rightarrow \mathit{Loc}\ \alpha \\
&\textbf{data family}\ \mathit{Ctx}\ (\phi :: \star \rightarrow \star) :: \star \rightarrow \star
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{enter} :: (\mathit{Regular}\ \alpha, \mathit{Zipper}\ (\mathit{PF}\ \alpha)) \Rightarrow \alpha \rightarrow \mathit{Loc}\ \alpha \\
&\mathit{leave} :: \mathit{Loc}\ \alpha \rightarrow \alpha \\
&\mathit{up}, \mathit{down}, \mathit{left}, \mathit{right} :: \mathit{Loc}\ \alpha \rightarrow \mathit{Maybe}\ (\mathit{Loc}\ \alpha) \\
&\mathit{on} \qquad\quad :: \mathit{Loc}\ \alpha \rightarrow \alpha \\
&\mathit{updateM} :: \mathit{Monad}\ \phi \Rightarrow (\alpha \rightarrow \phi\ \alpha) \rightarrow \mathit{Loc}\ \alpha \rightarrow \phi\ (\mathit{Loc}\ \alpha)
\end{aligned}
$$

1. Adding state to a zipper I

Now we embed this zipper in a state monad:

$$\textbf{type}\ \mathit{ZipperMonad}\ \alpha\ \beta = \mathit{StateT}\ (\mathit{Loc}\ \alpha)\ \mathit{Maybe}\ \beta$$

Note that $\mathit{ZipperMonad}\ \alpha\ \beta \approx \mathit{Loc}\ \alpha \rightarrow \mathit{Maybe}\ (\beta, \mathit{Loc}\ \alpha)$.

1. Adding state to a zipper I

Now we embed this zipper in a state monad:

$$\textbf{type}\ \mathit{ZipperMonad}\ \alpha\ \beta = \mathit{StateT}\ (\mathit{Loc}\ \alpha)\ \mathit{Maybe}\ \beta$$

Note that $\mathit{ZipperMonad}\ \alpha\ \beta \approx \mathit{Loc}\ \alpha \to \mathit{Maybe}\ (\beta, \mathit{Loc}\ \alpha)$.

Moving around:

$$\begin{aligned}
&\mathit{moveZS} :: (\mathit{Loc}\ \alpha \to \mathit{Maybe}\ (\mathit{Loc}\ \alpha)) \to \mathit{ZipperMonad}\ \alpha\ \alpha \\
&\mathit{moveZS}\ m = \mathit{StateT}\ \$\ \lambda l \to m\ l \gg\!= \lambda l' \to \mathit{return}\ (\mathit{on}\ l', l')
\end{aligned}$$

$$\begin{aligned}
&\mathit{upZS}, \mathit{downZS}, \mathit{leftZS}, \mathit{rightZS} :: \mathit{ZipperMonad}\ \alpha\ \alpha \\
&\mathit{upZS} \quad\;\; = \mathit{moveZS}\ \mathit{up} \\
&\mathit{downZS} = \mathit{moveZS}\ \mathit{down} \\
&\mathit{leftZS} \quad = \mathit{moveZS}\ \mathit{left} \\
&\mathit{rightZS} \;\; = \mathit{moveZS}\ \mathit{right}
\end{aligned}$$

1. Adding state to a zipper II

Updating the value at the current location:

$$
\begin{aligned}
&\mathit{updateZS} :: (\alpha \to \alpha) \to \mathit{ZipperMonad}\ \alpha\ \alpha \\
&\mathit{updateZS}\ f = \textbf{do}\ l \leftarrow \mathit{get} \\
&\qquad\qquad\qquad\quad\ \textbf{let}\ \mathit{Just}\ l' = \mathit{updateM}\ (\mathit{Just} \circ f)\ l \\
&\qquad\qquad\qquad\quad\ \mathit{put}\ l' \\
&\qquad\qquad\qquad\quad\ \mathit{return}\ (\mathit{on}\ l')
\end{aligned}
$$

1. Another example

And that's all that's needed!

Another example, this time a swap:

$$
\begin{aligned}
&\mathit{swap} :: \mathit{Maybe}\ \mathit{Expr}\\
&\mathit{swap} = \mathit{navigateZS}\ (\mathit{Add}\ (\mathit{Const}\ 1)\ (\mathit{Var}\ \texttt{"a"}))\ \$\\
&\quad \mathbf{do}\ l \leftarrow \mathit{downZS}\\
&\qquad r \leftarrow \mathit{rightZS}\\
&\qquad \mathit{updateZS}\ (\mathit{const}\ l)\\
&\qquad \mathit{leftZS}\\
&\qquad \mathit{updateZS}\ (\mathit{const}\ r)
\end{aligned}
$$

$$\mathit{swap} \equiv \mathit{Just}\ (\mathit{Add}\ (\mathit{Var}\ \texttt{"a"})\ (\mathit{Const}\ 1))$$

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

2. Rewriting

Our second approach uses a zipper to focus on a specific location, and rewrite rules to apply transformations:

$$\begin{aligned}
&\mathit{insert} :: \mathit{Maybe}\ \mathit{Expr}\\
&\mathit{insert} = \mathit{apply}\ [(\mathit{down} \ggg \mathit{right}, \mathit{addNeg})]\ ((\mathit{Add}\ (\mathit{Const}\ 1)\ (\mathit{Var}\ \texttt{"a"}))\\
&\quad \textbf{where}\ \mathit{addNeg} :: \mathit{Rule}\ \mathit{Expr}\\
&\qquad\qquad \mathit{addNeg} = \mathit{rule}\ \$\ \lambda x \to x : \leadsto : \mathit{Neg}\ x
\end{aligned}$$

$$\mathit{insert} \equiv \mathit{Just}\ (\mathit{Add}\ (\mathit{Const}\ 1)\ (\mathit{Neg}\ (\mathit{Var}\ \texttt{"a"})))$$

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:

$$
\begin{aligned}
&\mathbf{type}\ \mathit{Ext}\ \phi &&= \mathit{K}\ \mathit{Metavar}\ {:}{+}{:}\ \phi \\
&\mathbf{type}\ \mathit{Metavar} &&= \mathit{Int} \\
&\mathbf{data}\ \mu\,\phi &&= \mathit{In}\,(\phi\,(\mu\,\phi)) \\
&\mathbf{type}\ \mathit{Scheme}\ \phi &&= \mu\,(\mathit{Ext}\ \phi) \\
&\mathbf{type}\ \mathit{SchemeOf}\ \alpha &&= \mathit{Scheme}\,(\mathit{PF}\ \alpha)
\end{aligned}
$$

2. More examples

Deleting a node:

$$
\begin{aligned}
&\mathit{delete} :: \mathit{Maybe\ Expr}\\
&\mathit{delete} = \mathit{apply}\ [(\mathit{down} \ggg \mathit{right}, \mathit{removeNeg})]\ \mathit{expr}_2\ \textbf{where}\\
&\quad \mathit{removeNeg} :: \mathit{Rule\ Expr}\\
&\quad \mathit{removeNeg} = \mathit{rule}\,\$\,\lambda x \rightarrow \mathit{Neg}\ x :\leadsto: x
\end{aligned}
$$

2. More examples

Deleting a node:

$$\mathit{delete} :: \mathit{Maybe}\ \mathit{Expr}$$
$$\mathit{delete} = \mathit{apply}\ [(\mathit{down} \ggg \mathit{right}, \mathit{removeNeg})]\ \mathit{expr}_2\ \textbf{where}$$
$$\quad \mathit{removeNeg} :: \mathit{Rule}\ \mathit{Expr}$$
$$\quad \mathit{removeNeg} = \mathit{rule}\,\$\,\lambda x \rightarrow \mathit{Neg}\ x :\rightsquigarrow: x$$

And here is the swap example:

$$\mathit{swapExpr} :: \mathit{Rule}\ \mathit{Expr}$$
$$\mathit{swapExpr} = \mathit{rule}\,\$\,\lambda a\ b \rightarrow \mathit{Add}\ a\ b :\rightsquigarrow: \mathit{Add}\ b\ a$$

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:

$$
\begin{aligned}
&\mathit{expandedSwap} :: \mathit{Rule}\ \mathit{Expr} \\
&\mathit{expandedSwap} = \\
&\qquad\quad \mathit{In}\,(R\,(R\,(R\,(R \\
&\qquad\qquad (I\,(\mathit{In}\,(L\,(K\,0)))\ {:}{\times}{:}\ I\,(\mathit{In}\,(L\,(K\,1)))))))) \\
&\quad {:}{\leadsto}{:}\ \mathit{In}\,(R\,(R\,(R\,(R \\
&\qquad\qquad (I\,(\mathit{In}\,(L\,(K\,1)))\ {:}{\times}{:}\ I\,(\mathit{In}\,(L\,(K\,0))))))))
\end{aligned}
$$

3. Explicit encoding

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

$$
\begin{array}{l}
\mathit{insert} :: \mathit{Maybe\ Expr} \\
\mathit{insert} = \mathit{apply\ addNeg}\ (\mathit{Add}\ (\mathit{Const}\ 1)\ (\mathit{Var}\ \texttt{"a"}))\ \textbf{where} \\
\quad \mathit{addNeg} :: \mathit{Transformation\ Expr} \\
\quad \mathit{addNeg} = [([1], \mathit{Neg}\ (\mathit{Ref}\ [1]))]
\end{array}
$$

3. Explicit encoding

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

$$\begin{array}{l}
\mathit{insert} :: \mathit{Maybe}\ \mathit{Expr} \\
\mathit{insert} = \mathit{apply}\ \mathit{addNeg}\ (\mathit{Add}\ (\mathit{Const}\ 1)\ (\mathit{Var}\ \texttt{"a"}))\ \textbf{where} \\
\quad \mathit{addNeg} :: \mathit{Transformation}\ \mathit{Expr} \\
\quad \mathit{addNeg} = [([1], \mathit{Neg}\ (\mathit{Ref}\ [1]))]
\end{array}$$

$$\begin{array}{l}
\mathit{delete} :: \mathit{Maybe}\ \mathit{Expr} \\
\mathit{delete} = \mathit{apply}\ \mathit{delNeg}\ (\mathit{Add}\ (\mathit{Const}\ 1)\ (\mathit{Neg}\ (\mathit{Var}\ \texttt{"a"})))\ \textbf{where} \\
\quad \mathit{delNeg} :: \mathit{Transformation}\ \mathit{Expr} \\
\quad \mathit{delNeg} = [([1], \mathit{Ref}\ [1,0])]
\end{array}$$

3. Explicit encoding

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

$$
\begin{aligned}
&\mathit{insert} :: \mathit{Maybe\ Expr}\\
&\mathit{insert} = \mathit{apply\ addNeg}\ (\mathit{Add}\ (\mathit{Const}\ 1)\ (\mathit{Var}\ \texttt{"a"}))\ \textbf{where}\\
&\quad \mathit{addNeg} :: \mathit{Transformation\ Expr}\\
&\quad \mathit{addNeg} = [([1], \mathit{Neg}\ (\mathit{Ref}\ [1]))]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{delete} :: \mathit{Maybe\ Expr}\\
&\mathit{delete} = \mathit{apply\ delNeg}\ (\mathit{Add}\ (\mathit{Const}\ 1)\ (\mathit{Neg}\ (\mathit{Var}\ \texttt{"a"})))\ \textbf{where}\\
&\quad \mathit{delNeg} :: \mathit{Transformation\ Expr}\\
&\quad \mathit{delNeg} = [([1], \mathit{Ref}\ [1,0])]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{swap} :: \mathit{Maybe\ Expr}\\
&\mathit{swap} = \mathit{apply\ swap'}\ (\mathit{Add}\ (\mathit{Const}\ 1)\ (\mathit{Var}\ \texttt{"a"}))\ \textbf{where}\\
&\quad \mathit{swap'} :: \mathit{Transformation\ Expr}\\
&\quad \mathit{swap'} = [([0], \mathit{Ref}\ [1]), ([1], \mathit{Ref}\ [0])]
\end{aligned}
$$

3. Paths, references, and transformations

Paths are encoded as lists of integers:

$$\textbf{type}\ \mathit{Path} = [\,\mathit{Int}\,]$$

Yes, this is suboptimal.

3. Paths, references, and transformations

Paths are encoded as lists of integers:

$$\textbf{type}\ \mathit{Path} = [\,\mathit{Int}\,]$$

Yes, this is suboptimal.

References are encoded similarly to meta-variables for rewriting:

$$\begin{aligned}\textbf{data}\ \mathit{WithRef}\ \alpha\ \beta &= \mathit{InR}\ (\mathit{PF}\ \alpha\ \beta)\\ &\mid\ \mathit{Ref}\ \mathit{Path}\end{aligned}$$

3. Paths, references, and transformations

Paths are encoded as lists of integers:

$$\textbf{type}\ \mathit{Path} = [\,\mathit{Int}\,]$$

Yes, this is suboptimal.

References are encoded similarly to meta-variables for rewriting:

$$\begin{aligned}\textbf{data}\ \mathit{WithRef}\ \alpha\ \beta &= \mathit{InR}\ (\mathit{PF}\ \alpha\ \beta)\\ &\mid\ \mathit{Ref}\ \ \mathit{Path}\end{aligned}$$

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

$$\textbf{type}\ \mathit{Transformation}\ \alpha = [\,(\mathit{Path}, \mu\,(\mathit{WithRef}\ \alpha))\,]$$

3. Ease of use

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

$$\textbf{let}\ \mathit{expr}_1 = \mathit{Add}\ (\mathit{Const}\ 1)\ (\mathit{Var}\ \texttt{"a"})$$

$$\begin{array}{l}
\mathit{insert} :: \mathit{Maybe\ Expr} \\
\mathit{insert} = \mathit{apply}\ \mathit{addNeg}\ \mathit{expr}_1\ \textbf{where} \\
\quad \mathit{addNeg} :: \mathit{Transformation\ Expr} \\
\quad \mathit{addNeg} = [([1], \mathit{In} \circ \mathit{InR} \circ R \circ R \circ L \circ I \circ \mathit{In}\ \$\ \mathit{Ref}\ [1])]
\end{array}$$

3. Ease of use

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

$$\textbf{let}\ \mathit{expr_1} = \mathit{Add}\ (\mathit{Const}\ 1)\ (\mathit{Var}\ \texttt{"a"})$$

$$\begin{aligned}
&\mathit{insert} :: \mathit{Maybe\ Expr}\\
&\mathit{insert} = \mathit{apply}\ \mathit{addNeg}\ \mathit{expr_1}\ \textbf{where}\\
&\quad \mathit{addNeg} :: \mathit{Transformation\ Expr}\\
&\quad \mathit{addNeg} = [([1], \mathit{In} \circ \mathit{InR} \circ \mathit{R} \circ \mathit{R} \circ \mathit{L} \circ \mathit{I} \circ \mathit{In}\ \$\ \mathit{Ref}\ [1])]
\end{aligned}$$

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

$$\begin{aligned}
&\mathit{insert} :: \mathit{Maybe\ Expr}\\
&\mathit{insert} = \mathit{apply}\ (\mathit{fromTransformation_{EH}}\ \mathit{addNeg})\ \mathit{expr_1}\ \textbf{where}\\
&\quad \mathit{addNeg} :: \mathit{Transformation_{EH}}\ \mathit{Expr}\\
&\quad \mathit{addNeg} = [([1], \mathit{Neg_{EH}}\ (\mathit{Ref_{EH}}\ [1]))]
\end{aligned}$$

3. Generic diff

Unlike the previous approaches, this time we have a
$\mathit{diff} :: (\mathit{Regular}\ \alpha, \ldots) \Rightarrow \alpha \rightarrow \alpha \rightarrow \mathit{Transformation}\ \alpha$ such that:

$$\forall a, b.\ \mathit{apply}\ (\mathit{diff}\ \ a\ b)\ a \equiv \mathit{Just}\ b$$

Our *diff* returns *Transformation*s 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

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`.

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`.

Questions?