Universiteit Utrecht

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

Typed Transformations of Typed Abstract Syntax

Arthur Baars   Doaitse Swierstra   Marcos Viera

Instituto Tecnológico de Informática, Universidad Politécnica de Valencia, Spain
Dept. of Information and Computing Sciences, Utrecht University, the Netherlands
Instituto de Computación, Universidad de la República, Uruguay

Lecture14, AFP, Jan 17, 2011

1. Why we need typed abstract syntax?

What is typed abstract syntax?

- values have types
- values can be composed
- types prevent invalid compositions of values

What is typed abstract syntax?

- descriptions of values have types
- descriptions of values can be composed
- types prevent invalid compositions of descriptions of values

$$
\begin{aligned}
&\textbf{data}\ \mathit{Expr}\ a\ \textbf{where}\\
&\qquad \mathit{Val} \ \ :: a \to \mathit{Expr}\ a\\
&\qquad \mathit{Apply} :: \mathit{Expr}\ (b \to a) \to (\mathit{Expr}\ b) \to \mathit{Expr}\ a
\end{aligned}
$$

Where does Typed Astract Syntax arise?

- we want to implement *Embedded Domain Specific Languages*

Our ultimate goal is to "compile" embedded languages just as we compile normal languages.

Where does Typed Astract Syntax arise? §1

- we want to implement *Embedded Domain Specific Languages*
- which inherit their type system from the host language

Our ultimate goal is to "compile" embedded languages just as we compile normal languages.

Where does Typed Astract Syntax arise?

- we want to implement *Embedded Domain Specific Languages*
- which inherit their type system from the host language
- instead of directly building the semantics we:

Our ultimate goal is to “compile” embedded languages just as we compile normal languages.

Where does Typed Astract Syntax arise?

- we want to implement *Embedded Domain Specific Languages*
- which inherit their type system from the host language
- instead of directly building the semantics we:
  - build the typed abstract syntax tree

Our ultimate goal is to "compile" embedded languages just as we compile normal languages.

Where does Typed Astract Syntax arise?

- we want to implement *Embedded Domain Specific Languages*
- which inherit their type system from the host language
- instead of directly building the semantics we:
  - build the typed abstract syntax tree
  - which we *analyse*, *transform* and from which we finally construct the *semantics*

Our ultimate goal is to “compile” embedded languages just as we compile normal languages.

GADTs

Generalised Algebraic Data Types enable us to encode the typing of the EDSL in the typing of the host language:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{data}\ \mathit{Exp}\ a\ \textbf{where}} \\
\quad \mathit{IntVal} & :: \mathit{Int} & & \to \mathit{Exp}\ \mathit{Int} \\
\quad \mathit{BoolVal} & :: \mathit{Bool} & & \to \mathit{Exp}\ \mathit{Bool} \\
\quad \mathit{Add} & :: \mathit{Exp}\ \mathit{Int} & \to \mathit{Exp}\ \mathit{Int} & \to \mathit{Exp}\ \mathit{Int} \\
\quad \mathit{Cons1} & :: \mathit{Exp}\ a & \to \mathit{Exp}\ [a] & \to \mathit{Exp}\ [a] \\
\quad \mathit{Nil1} & :: & & \mathit{Exp}\ [a] \\
\quad \mathit{LessThan} & :: \mathit{Exp}\ \mathit{Int} & \to \mathit{Exp}\ \mathit{Int} & \to \mathit{Exp}\ \mathit{Bool} \\
\quad \mathit{If} & :: \mathit{Exp}\ \mathit{Bool} & \to \mathit{Exp}\ a & \\
 & & \to \mathit{Exp}\ a & \to \mathit{Exp}\ a
\end{array}
$$

GADTs

Generalised Algebraic Data Types enable us to encode the typing of the EDSL in the typing of the host language:

$$
\begin{array}{llllll}
\multicolumn{6}{l}{\textbf{data}\ \mathit{Exp}\ a\ \textbf{where}} \\
\quad \mathit{IntVal} & :: \mathit{Int} & & & \rightarrow & \mathit{Exp}\ \mathit{Int} \\
\quad \mathit{BoolVal} & :: \mathit{Bool} & & & \rightarrow & \mathit{Exp}\ \mathit{Bool} \\
\quad \mathit{Add} & :: \mathit{Exp}\ \mathit{Int} & \rightarrow & \mathit{Exp}\ \mathit{Int} & \rightarrow & \mathit{Exp}\ \mathit{Int} \\
\quad \mathit{Cons1} & :: \mathit{Exp}\ a & \rightarrow & \mathit{Exp}\ [a] & \rightarrow & \mathit{Exp}\ [a] \\
\quad \mathit{Nil1} & :: & & & & \mathit{Exp}\ [a] \\
\quad \mathit{LessThan} & :: \mathit{Exp}\ \mathit{Int} & \rightarrow & \mathit{Exp}\ \mathit{Int} & \rightarrow & \mathit{Exp}\ \mathit{Bool} \\
\quad \mathit{If} & :: \mathit{Exp}\ \mathit{Bool} & \rightarrow & \mathit{Exp}\ a & & \\
 & & \rightarrow & \mathit{Exp}\ a & \rightarrow & \mathit{Exp}\ a
\end{array}
$$

The price we pay is that we have to maintain well-typedness during program transformations.

EDSL's may contain references

We extend *Expr* with an argument describing the environment in which referred values are located:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Expr}\ a\ \mathit{env}\ \textbf{where}\\
&\quad \mathit{Var} :: \mathit{Ref}\ a\ \mathit{env} \to \mathit{Expr}\ a\ \mathit{env}\\
&\quad \mathit{IntVal} :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int}\ \mathit{env}\\
&\quad \mathit{BoolVal} :: \mathit{Bool} \to \mathit{Expr}\ \mathit{Bool}\ \mathit{env}\\
&\quad \ldots
\end{aligned}
$$

EDSL's may contain references

We extend *Expr* with an argument describing the environment in which referred values are located:

```
data Expr a env where
      Var     :: Ref a env → Expr a env
      IntVal  :: Int       → Expr Int env
      BoolVal :: Bool      → Expr Bool env
      ...
```

```
lookup :: Ref  a env → env → a
eval   :: Expr a env → env → a
eval (Var r)       e = lookup r e
eval (IntVal i)    _ = i
eval (BoolVal b)   _ = b
eval (Add x y)     e = eval x e + eval y e
...
```

Sidestepping: Type equality

Using a GADT we can provide the witness of the proof that two types are equal:

$$\textbf{data}\ \mathit{Equal} :: * \rightarrow * \rightarrow * \ \textbf{where}$$
$$\quad \mathit{Eq} :: \mathit{Equal}\ a\ a$$

Sidestepping: Type equality

Using a GADT we can provide the witness of the proof that two types are equal:

$$\textbf{data}\ \mathit{Equal} :: * \rightarrow * \rightarrow * \ \textbf{where}$$
$$\quad \mathit{Eq} :: \mathit{Equal}\ a\ a$$

If a non-$\bot$ value $\mathit{Eq}\ a\ b$ takes part in a successful pattern match, the type checker may conclude that the types a and b are the same; otherwise the Eq could not have been produced.

Typed References

Ref-erences are labelled with the type a of the value they point to in an environment *env*:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Ref}\ a\ \mathit{env}\ \textbf{where} \\
&\quad \mathit{Zero} :: \qquad\qquad\qquad\ \mathit{Ref}\ a\ (\mathit{env}', a) \\
&\quad \mathit{Suc}\ \ :: \mathit{Ref}\ a\ \mathit{env}' \to \mathit{Ref}\ a\ (\mathit{env}', b)
\end{aligned}
$$

Typed References

Environments are nested products

Ref-erences are labelled with the type *a* of the value they point to in an environment *env*:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Ref}\ a\ \mathit{env}\ \textbf{where}\\
&\quad \mathit{Zero} :: \qquad\qquad\qquad\ \mathit{Ref}\ a\ (\mathit{env}', a)\\
&\quad \mathit{Suc}\ \ :: \mathit{Ref}\ a\ \mathit{env}' \to \mathit{Ref}\ a\ (\mathit{env}', b)
\end{aligned}
$$

Typed References

Ref-erences are labelled with the type a of the value they point to in an environment env:

$$
\begin{array}{l}
\mathbf{data}\ Ref\ a\ env\ \mathbf{where} \\
\quad Zero :: \qquad\qquad\qquad Ref\ a\ (env', a) \\
\quad Suc\ \ :: Ref\ a\ env' \rightarrow Ref\ a\ (env', b)
\end{array}
$$

References can be compared; if they are equal they return the proof that the values they refer to have the same type:

$$
\begin{array}{l}
match :: Ref\ a\ env \rightarrow Ref\ b\ env \rightarrow Maybe\ (Equal\ a\ b) \\
match\ Zero\ \quad Zero \quad\ = Just\ Eq \\
match\ (Suc\ x)\ (Suc\ y) = match\ x\ y \\
match\ \_ \qquad\quad \_ \qquad\quad = Nothing
\end{array}
$$

Typed References

Ref-erences are labelled with the type a of the value they point to in an environment env:

$$
\begin{aligned}
&\textbf{data}\ Ref\ a\ env\ \textbf{where}\\
&\quad Zero :: \qquad\qquad\qquad\ Ref\ a\ (env', a)\\
&\quad Suc\ :: Ref\ a\ env' \to Ref\ a\ (env', b)
\end{aligned}
$$

References can be compared; if they are equal they return the proof that the values they refer to have the same type:

$$
\begin{aligned}
&match :: Ref\ a\ env \to Ref\ b\ env \to Maybe\ (Equal\ a\ b)\\
&match\ Zero\qquad Zero\qquad = Just\ Eq\\
&match\ (Suc\ x)\ (Suc\ y) = match\ x\ y\\
&match\ \_\qquad\quad \_\qquad\quad\ = Nothing
\end{aligned}
$$

Here a and b are equal

Mutally Recursive Declarative Structures

We want to represent:

$$\textbf{let}\ x = 1 : y \\ \quad\ \ y = 2 : x$$

A first attempt:

$$\textbf{type}\ \mathit{TwoLists} = (((), \mathit{Expr}\ [\mathit{Int}]\ \mathit{TwoLists}) \\ , \mathit{Expr}\ [\mathit{Int}]\ \mathit{TwoLists})$$

Mutally Recursive Declarative Structures

We want to represent:

$$\textbf{let } x = 1 : y \\ \quad\;\; y = 2 : x$$

A first attempt:

$$\textbf{type } \mathit{TwoLists} = (((), \mathit{Expr}\ [\mathit{Int}]\ \mathit{TwoLists}) \\ \quad , \mathit{Expr}\ [\mathit{Int}]\ \mathit{TwoLists})$$

Unfortunately this is not correct Haskell: the type is recursive

Mutally Recursive Declarative Structures

We want to represent:

$$\textbf{let}\ x = 1 : y \\ \quad\ y = 2 : x$$

We split the environment in two type parameters: the *used* environment and the *defined* environment:

$$\begin{aligned}&\textbf{data}\ \mathit{Env} :: (\ * \ \to \ * \ \to \ * \ ) \to \ * \ \to \ * \ \to \ * \\ &\quad \textbf{where}\ \mathit{Empty} :: \mathit{Env}\ \mathit{term}\ \mathit{used}\ () \\ &\qquad\qquad \mathit{Ext} \quad :: \mathit{Env}\ \mathit{term}\ \mathit{used}\ \mathit{defined} \to \mathit{term}\ a\ \mathit{used} \\ &\qquad\qquad\qquad\quad \to \mathit{Env}\ \mathit{term}\ \mathit{used}\ (\mathit{defined}, a)\end{aligned}$$

Mutally Recursive Declarative Structures

We want to represent:

$$\begin{aligned} \textbf{let } x &= 1 : y \\ y &= 2 : x \end{aligned}$$

We split the environment in two type parameters: the *used* environment and the *defined* environment:

$$\begin{aligned} &\textbf{data } Env :: (\ * \ \to \ * \ \to \ * \ ) \to \ * \ \to \ * \ \to \ * \\ &\quad \textbf{where } Empty :: Env\ term\ used\ () \\ &\qquad\qquad\ \ Ext \quad :: Env\ term\ used\ defined \to term\ a\ used \\ &\qquad\qquad\qquad\qquad \to Env\ term\ used\ (defined, a) \end{aligned}$$

By choosing the two environment parameters to be the same we enforce that the environemnt is closed.

Example

The expression:

$$\begin{aligned} \textbf{let } x &= 1 : y \\ y &= 2 : x \end{aligned}$$

is now encoded as:

$$\begin{array}{lll} \textbf{type } \mathit{Final} = (((), [\mathit{Int}]), [\mathit{Int}]) \\ x & = & \mathit{Var}\ (\mathit{Suc}\ \mathit{Zero}) :: \mathit{Expr}\ [\mathit{Int}]\ \mathit{Final} \\ y & = & \mathit{Var}\ \qquad \mathit{Zero} \ :: \mathit{Expr}\ [\mathit{Int}]\ \mathit{Final} \\ \mathit{decls} & :: & \mathit{Env}\ \mathit{Expr}\ \mathit{Final}\ \mathit{Final} \\ \mathit{decls} & = & \mathit{Empty}\ \text{`}\mathit{Ext}\text{`}\ \mathit{Cons}\ (\mathit{IntVal}\ 1)\ y \\ & & \qquad\quad \text{`}\mathit{Ext}\text{`}\ \mathit{Cons}\ (\mathit{Intval}\ 2)\ x \end{array}$$

Example

The expression:

We have nicer syntax for this

$$\textbf{let } x = 1 : y \\ \quad\; y = 2 : x$$

is now encoded as:

$$\begin{array}{lll} \textbf{type } \mathit{Final} & = & (((),[\mathit{Int}]),[\mathit{Int}]) \\ x & = & \mathit{Var}\ (\mathit{Suc}\ \mathit{Zero}) :: \mathit{Expr}\ [\mathit{Int}]\ \mathit{Final} \\ y & = & \mathit{Var}\ \mathit{Zero} :: \mathit{Expr}\ [\mathit{Int}]\ \mathit{Final} \\ \mathit{decls} & :: & \mathit{Env}\ \mathit{Expr}\ \mathit{Final}\ \mathit{Final} \\ \mathit{decls} & = & \mathit{Empty}\ \text{`}\mathit{Ext}\text{`}\ \mathit{Cons}\ (\mathit{IntVal}\ 1)\ y \\ & & \quad\quad \text{`}\mathit{Ext}\text{`}\ \mathit{Cons}\ (\mathit{Intval}\ 2)\ x \end{array}$$

The problem: Common Subexpression Elimination

Suppose we want to transform the program:

$$a = 4;$$
$$b = (a + 4) + (a + 4);$$

into:

$$a = 4;$$
$$x = a + a;$$
$$b = x + x;$$

The problem: Common Subexpression Elimination

Suppose we want to transform the program:

$$a = 4;$$
$$b = (a + 4) + (a + 4);$$

into:

$$a = 4;$$
$$x = a + a;$$
$$b = x + x;$$

In order to do so we have to build a new environment, containing the extra definition for x, and the new right hand sides for a and b. This new environment is built incrementally.

The Transformation Library

Eventually all references have to point into the final environment. We thus introduce the following types:

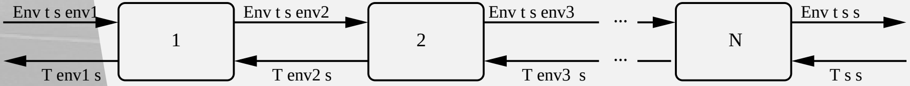
$$\textbf{type}\ \mathit{FinalEnv}\ t\ \mathit{usedef} = \mathit{Env}\ t\ \mathit{usedef}\ \mathit{usedef}$$
$$\textbf{newtype}\ T\ e\ s = T\ \{\mathit{unT} :: \forall x\ .\ \mathit{Ref}\ x\ e \rightarrow \mathit{Ref}\ x\ s\}$$

The Transformation Library

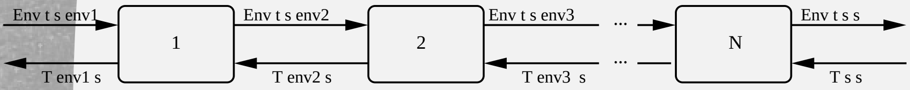
Eventually all references have to point into the final environment. We thus introduce the following types:

$$\textbf{type}\ FinalEnv\ t\ usedef = Env\ t\ usedef\ usedef$$
$$\textbf{newtype}\ T\ e\ s = T\ \{unT :: \forall x\ .\ Ref\ x\ e \rightarrow Ref\ x\ s\}$$

Whenever we add a new element to the environment under construction we have to update the already existing references. Instead we make a function available which maps them directly into the final environment:



The *Trafo* type



$$
\begin{aligned}
&\textbf{type } \mathit{Trafo}\ t\ s = \\
&\quad \forall \mathit{env1}\ .\ \exists \mathit{env2}\ .\qquad T\ \mathit{env2}\ s \to \mathit{Env}\ t\ s\ \mathit{env1} \\
&\qquad\qquad\qquad\qquad\quad \to (T\ \mathit{env1}\ s,\quad \mathit{Env}\ t\ s\ \mathit{env2})
\end{aligned}
$$

$Env\ t\ s\ env1$ the environment constructed thus far

$T\ env2\ s$ represents the number of future additions to the environment

$Env\ t\ s\ env2$ the updated environment, in which $env2$ is (an extension of) $env1$

$T\ env1\ s$ the updated $T\ env2\ s$

Extend with an arrow like interface

We extend the type with an arrow-like interface:

$$\mathbf{type}\ \mathit{Trafo}\ t\ s\ a\ b = \forall \mathit{env1}\ .\ \exists \mathit{env2}\ .\ a \rightarrow T\ \mathit{env2}\ s \rightarrow \mathit{Env}\ t\ s\ \mathit{env1} \rightarrow (b, T\ \mathit{env1}\ s, \mathit{Env}\ t\ s\ \mathit{env2})$$

In our example a will e.g. be the mapping which tells us where the old variables have ended up in the new environment.

Meta-information

Since the elements in the constructed environment cannot be fully inspected (parts depend on the T which is still has to be constructed by future transformations), we maintain meta information m:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Trafo}\ m\ t\ s\ a\ b = \\
&\quad \forall \mathit{env1}\ .\ m\ \mathit{env1} \\
&\qquad \to \exists \mathit{env2}\ . \\
&\qquad\quad (\ m\ \mathit{env2} \\
&\qquad\quad ,\ \ a \to T\ \mathit{env2}\ s \to \mathit{Env}\ t\ s\ \mathit{env1} \\
&\qquad\qquad \to (b,\ T\ \mathit{env1}\ s,\ \mathit{Env}\ t\ s\ \mathit{env2}) \\
&\qquad\quad)
\end{aligned}
$$

Haskelize

Since Haskell only allows existenstial constructors in combination with a data constructor we have to write:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Trafo}\ m\ t\ s\ a\ b = \\
\quad \mathit{Trafo}\ (\forall \mathit{env1}\ .\ m\ \mathit{env1} \to \mathit{TrafoE}\ m\ t\ s\ a\ b\ \mathit{env1}) \\
\textbf{data}\ \mathit{TrafoE}\ m\ t\ s\ a\ b\ \mathit{env1} = \\
\quad \forall \mathit{env2}\ .\ \mathit{TrafoE}\ (m\ \mathit{env2}) \\
\qquad\qquad\qquad (\quad a \to T\ \mathit{env2}\ s \to \mathit{Env}\ t\ s\ \mathit{env1} \\
\qquad\qquad\qquad\quad \to (b,\ T\ \mathit{env1}\ s,\ \mathit{Env}\ t\ s\ \mathit{env2}) \\
\qquad\qquad\qquad)
\end{array}
$$

Creating a new Ref-erence

The meta-data type has to be filled in depending on the situation:

$$newSRef :: Trafo\ Unit\ t\ s\ (t\ a\ s)\ (Ref\ a\ s)$$
$$\mathbf{data}\ Unit\ s = Unit$$

Here $t\ a\ s$ is the term we add to the environemnt, and $Ref\ a\ s$ is the reference pointing to this element in the final environment!

Creating a new *Ref*-erence

$$newSRef = Trafo\ (\lambda\_ \rightarrow TrafoE\ Unit\ extEnv)$$

$$
\begin{aligned}
extEnv &:: t\ a\ s \rightarrow T\ (e, a)\ s \rightarrow Env\ t\ s\ e \\
&\quad \rightarrow (Ref\ a\ s, T\ e\ s, Env\ t\ s\ (e, a)) \\
extEnv & \\
&= \lambda ta\ (T\ tr)\ env \rightarrow (tr\ Zero, T\ (tr\ .\ Suc), Ext\ env\ ta)
\end{aligned}
$$

Creating a new *Ref*-erence

$$newSRef = Trafo\ (\lambda\_ \to TrafoE\ Unit\ extEnv)$$

$$\begin{aligned} extEnv &:: \quad t\ a\ s \to T\ (e, a)\ s \to Env\ t\ s\ e \\ &\qquad \to (Ref\ a\ s, T\ e\ s, Env\ t\ s\ (e, a)) \\ extEnv & \\ &= \lambda ta\ (T\ tr)\ env \to (tr\ Zero, T\ (tr\ .\ Suc), Ext\ env\ ta) \end{aligned}$$

Tell the predecessors that an element was added

Creating a new *Ref*-erence

$$newSRef = Trafo\ (\lambda\_ \rightarrow TrafoE\ Unit\ extEnv)$$

$$\begin{aligned} extEnv :: &\ t\ a\ s \rightarrow T\ (e, a)\ s \rightarrow Env\ t\ s\ e \\ &\rightarrow (Ref\ a\ s, T\ e\ s, Env\ t\ s\ (e, a)) \\ extEnv & \\ &= \lambda ta\ (T\ tr)\ env \rightarrow (tr\ Zero, T\ (tr\ .\ Suc), Ext\ env\ ta) \end{aligned}$$

Map the newly added element into the final environment

Tell the predecessors that an element was added

Creating a new *Ref*-erence


```
newSRef = Trafo (λ_ → TrafoE Unit extEnv)
```

Extend the environment with the new term *ta*


```
extEnv ::   t a s → T (e, a) s → Env t s e
         → (Ref a s, T e     s,   Env t s (e, a))
extEnv
  = λta (T tr) env → (tr Zero, T (tr . Suc), Ext env ta)
```

Map the newly added element into the final environment

Tell the predecessors that an element was added

Running the *Trafos*

When we are done we require that the used and the built environment are equally labelled, hence we use $FinalEnv$:

$$\textbf{data}\ Result\ m\ t\ b = \forall env2\ .\ Result\ (m\ env2)\ (b\ env2)\ (FinalEnv\ t\ env2)$$

Running the *Trafo*s

When we are done we require that the used and the built environment are equally labelled, hence we use *FinalEnv*:

$$\textbf{data}\ Result\ m\ t\ b$$
$$\quad = \forall env2\ .\ Result\ (m\ env2)\ (b\ env2)\ (FinalEnv\ t\ env2)$$

$$runTrafo :: \forall m\ t\ a\ b\ .\ (\forall s\ .\ Trafo\ m\ t\ s\ a\ (b\ s)) \to m\ () \to a \to Result\ m\ t\ b$$
$$runTrafo\ trafo\ m\ a =$$
$$\quad \textbf{let}\ Trafo\ trf = trafo$$
$$\quad\quad TrafoE\ m2\ f = trf\ m$$
$$\quad \textbf{in}\ \ \textbf{case}\ f\ a\ (T\ id)\ Empty\ \textbf{of}$$
$$\quad\quad (b, \_, env) \to Result\ m2\ b\ env$$

Running the *Trafo*s

When we are done we require that the used and the built environment are equally labelled, hence we use *FinalEnv*:

$$\begin{aligned}&\mathbf{data}\ \mathit{Result}\ m\ t\ b\\ &\quad = \forall \mathit{env2}\ .\ \mathit{Result}\ (m\ \mathit{env2})\ (b\ \mathit{env2})\\ &\qquad\qquad\qquad\quad (\mathit{FinalEnv}\ t\ \mathit{env2})\end{aligned}$$

$$\begin{aligned}&\mathit{runTrafo} :: \forall m\ t\ a\ b\ .\ (\forall s\ .\ \mathit{Trafo}\ m\ t\ s\ a\ (b\ s))\\ &\qquad\qquad\qquad \to m\ () \to a \to \mathit{Result}\ m\ t\ b\\ &\mathit{runTrafo}\ \mathit{trafo}\ m\ a =\\ &\quad \mathbf{let}\ \mathit{Trafo}\ \mathit{trf} \quad = \mathit{trafo}\\ &\qquad\ \ \mathit{TrafoE}\ m2\ f = \mathit{trf}\ m\\ &\quad \mathbf{in}\ \ \mathbf{case}\ f\ a\ (T\ \mathit{id})\ \mathit{Empty}\ \mathbf{of}\\ &\qquad\quad (b, \_, \mathit{env}) \to \mathit{Result}\ m2\ b\ \mathit{env}\end{aligned}$$

use equals *def*

Common Subexpression Elimination

After CSE we have a larger, closed environment:

$$\textbf{type}\ \mathit{Decls}\ \mathit{env}' = \mathit{Env}\ \mathit{Expr}\ \mathit{env}'\ \mathit{env}'$$

Common Subexpression Elimination

After CSE we have a larger, closed environment:

$$\textbf{type}\ \mathit{Decls}\ \mathit{env}' = \mathit{Env}\ \mathit{Expr}\ \mathit{env}'\ \mathit{env}'$$

We also compute a ref-transformer which maps old references in env to new references in env':

$$\textbf{data}\ \mathit{TDecls}\ \mathit{env} = \forall \mathit{env}'\ .\ \mathit{TDecls}\ (\mathit{Decls}\ \mathit{env}')\ (T\ \mathit{env}\ \mathit{env}')$$

Common Subexpression Elimination

After CSE we have a larger, closed environment:

$$\textbf{type}\ \mathit{Decls}\ \mathit{env}' = \mathit{Env}\ \mathit{Expr}\ \mathit{env}'\ \mathit{env}'$$

We also compute a ref-transformer which maps old references in env to new references in env':

$$\begin{aligned}\textbf{data}\ \mathit{TDecls}\ \mathit{env} = \forall \mathit{env}'\ .\ \mathit{TDecls}\ &(\mathit{Decls}\ \mathit{env}')\\ &(T\ \mathit{env}\ \mathit{env}')\end{aligned}$$

The type of cse now becomes:

$$\mathit{cse} :: \mathit{Decls}\ \mathit{env} \rightarrow \mathit{TDecls}\ \mathit{env}$$

Maintain a *Memo* table

In the meta-information we maintain a memo table, which we use to remember which expressions labelled with env have already been incorporated in the new environment:

$$
\begin{aligned}
&\mathbf{newtype}\ Memo\ env\ env' \\
&\quad = Memo \\
&\qquad (\forall x\ .\ Expr\ x\ env \\
&\qquad\qquad \to Maybe\ (Ref\ x\ env') \\
&\qquad) \\
&emptyMemo :: Memo\ env\ () \\
&emptyMemo = Memo\ (const\ Nothing)
\end{aligned}
$$

Maintain a *Memo* table

In the meta-information we maintain a memo table, which we use to remember which expressions labelled with *env* have already been incorporated in the new environment:

And we construct the type of our transformations:

$$
\begin{aligned}
&\mathbf{type}\ \mathit{TrafoCSE}\ \mathit{env} = \mathit{Trafo}\ (\mathit{Memo}\ \mathit{env})\ \mathit{Expr}\\
&\mathit{extMemo} :: \mathit{Expr}\ a\ \mathit{env} \to \mathit{Memo}\ \mathit{env}\ \mathit{env}'\\
&\qquad\qquad \to \mathit{Memo}\ \mathit{env}\ (\mathit{env}', a)\\
&\mathit{extMemo}\ e\ (\mathit{Memo}\ m)\\
&\qquad\qquad = \mathit{Memo}\ (\lambda s \to \mathbf{case}\ \mathit{equals}\ e\ s\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad \mathit{Just}\ \mathit{Eq} \to \mathit{Just}\ \mathit{Zero}\\
&\qquad\qquad\qquad\qquad \mathit{Nothing} \to \mathit{fmap}\ \mathit{Suc}\ (m\ s)\\
&\qquad\qquad\quad)
\end{aligned}
$$

Dealing with a single expression

$$
\begin{aligned}
&app\_cse :: Expr\ a\ env \\
&\qquad \rightarrow TrafoCSE\ env\ s\ (T\ env\ s)\ (Ref\ a\ s) \\
&app\_cse\ (Var\ r) = \mathbf{proc}\ (T\ tenv\_s) \rightarrow \\
&\qquad\qquad returnA \prec tenv\_s\ r
\end{aligned}
$$

Dealing with a single expression

$$
\begin{aligned}
&app\_cse :: Expr\ a\ env \\
&\qquad \rightarrow TrafoCSE\ env\ s\ (T\ env\ s)\ (Ref\ a\ s) \\
&app\_cse\ (Var\ r) = \mathbf{proc}\ (T\ tenv\_s) \rightarrow \\
&\qquad\qquad returnA \prec tenv\_s\ r
\end{aligned}
$$

Where do the old variables go?

Dealing with a single expression


```
app_cse :: Expr a env
           → TrafoCSE env s (T env s) (Ref a s)
app_cse (Var r) = proc (T tenv_s) →
                      returnA ≺ tenv_s r
```

Position of this expression

Where do the old variables go?

Dealing with a single expression

$$
\begin{aligned}
&app\_cse :: Expr\ a\ env \\
&\qquad \rightarrow TrafoCSE\ env\ s\ (T\ env\ s)\ (Ref\ a\ s) \\
&app\_cse\ (Var\ r) = \mathbf{proc}\ (T\ tenv\_s) \rightarrow \\
&\qquad\qquad returnA \prec tenv\_s\ r
\end{aligned}
$$

$$
\begin{aligned}
&app\_cse\ e@(LessThan\ x\ y) \\
&\quad = \mathbf{proc}\ tt \rightarrow \\
&\qquad \mathbf{do}\ l \leftarrow app\_cse\ x \prec tt \\
&\qquad\quad r \leftarrow app\_cse\ y \prec tt \\
&\qquad\quad insertIfNew\ e \prec LessThan\ (Var\ l)\ (Var\ r) \\
&\quad \ldots
\end{aligned}
$$

Running the transformations

$$refTransformer :: Env\ Ref\ s\ env \rightarrow T\ env\ s$$
$$refTransformer\ refs = T\ (\lambda r \rightarrow lookupEnv\ r\ refs)$$

Running the transformations

$$
\begin{array}{l}
\mathit{refTransformer} :: \mathit{Env}\ \mathit{Ref}\ s\ \mathit{env} \rightarrow T\ \mathit{env}\ s \\
\mathit{refTransformer}\ \mathit{refs} = T\ (\lambda r \rightarrow \mathit{lookupEnv}\ r\ \mathit{refs})
\end{array}
$$

The result of $\mathit{cse\_env}$ is used to compute its own input.Hence we use **mdo**:

$$
\begin{array}{l}
\mathit{trafo} :: \mathit{Decls}\ \mathit{env} \rightarrow \mathit{TrafoCSE}\ \mathit{env}\ s\ ()\ (T\ \mathit{env}\ s) \\
\mathit{trafo}\ \mathit{decls} = \mathbf{proc}\ \_ \rightarrow \\
\qquad\qquad \mathbf{mdo}\ \mathbf{let}\ \mathit{tt} = \mathit{refTransformer}\ \mathit{refs} \\
\qquad\qquad\qquad \mathit{refs} \leftarrow \mathit{cse\_env}\ \mathit{decls} \prec \mathit{tt} \\
\qquad\qquad\qquad \mathit{returnA} \prec \mathit{tt}
\end{array}
$$

Running the transformations


```
refTransformer :: Env Ref s env → T env s
refTransformer refs = T (λr → lookupEnv r refs)
```

The result of *cse_env* is used to compute its own input.Hence we use **mdo**:

```
trafo :: Decls env → TrafoCSE env s () (T env s)
trafo decls = proc _ →
                mdo let tt = refTransformer refs
                    refs ← cse_env decls ≺ tt
                    returnA ≺ tt
```

Finally we present the function *cse* which simply runs the *trafo* and extracts the result:

```
cse :: ∀env . Decls env → TDecls env
cse decls
  = case runTrafo (trafo decls) emptyMemo () of
        Result _ t env → TDecls env t
```

GHC problems

Unfortunately we have used lazy pattern binding on the existential type *TrafoE*:

```
data Trafo m t s a b =
   Trafo (∀env1 . m env1 → TrafoE m t s a b env1)
data TrafoE m t s a b env1 =
  ∀env2 . TrafoE (m env2)
                 (a → T env2 s → Env t s env1 →
                 (b,    T env1 s,    Env t s env2)
                 )
```

```
runTrafo :: ∀m t a b . (∀s . Trafo m t s a (b s))
                         → m () → a → Result m t b
runTrafo trafo m a =
   let Trafo trf       = trafo
       TRafoE m2 f = trf m
   in  case f a (T id) Empty of
         (b, _, env) → Result m2 b env
```

The blunt solution: *unsafeCoerce* §1

```
unsafeCoerce :: a → b
runTrafo  :: (∀s . Trafo m t s a (b s)) → m () → a
              → Result m t b
runTrafo trafo m a = case trafo of
  Trafo trf → case trf m of
    TrafoE m2 f →
      case f a (T unsafeCoerce) Empty of
        (rb, tt, env2) →
          Result (unsafeCoerce m2)
                 rb
                 (unsafeCoerce env2)
```

2. Conclusion

Why is this so complicated ...

If we have lazy evaluation, we also want it at the type level!

$$
\begin{array}{l}
f :: \forall a\ .\ (a \rightarrow \exists b\ (b, a, b \rightarrow b \rightarrow \mathit{Int})) \\
\mathbf{let}\ (b, a, g) = f\ \ b \\
\mathbf{in}\ \ g\ \ b\ \ a
\end{array}
$$

Why is this so complicated ...

If we have lazy evaluation, we also want it at the type level!

$$f :: \forall a \,.\, (a \rightarrow \exists b\ (b, a, b \rightarrow b \rightarrow Int))$$
$$\mathbf{let}\ (b, a, g) = f\ \ b$$
$$\mathbf{in}\ \ g\ b\ a$$

But this is not System-F!

Alternative: move the final s inwards

$$
\begin{array}{l}
\textbf{data}\ \mathit{Trafo2}\ m\ t\ a\ b = \\
\quad \mathit{Trafo2}\ (\forall \mathit{env1}\ .\ m\ \mathit{env1} \rightarrow \mathit{TrafoE2}\ m\ t\ a\ b\ \mathit{env1}) \\
\textbf{data}\ \mathit{TrafoE2}\ m\ t\ a\ b\ \mathit{env1} = \\
\quad \forall \mathit{env2}\ .\ \mathit{TrafoE2} \\
\qquad\qquad (m\ \mathit{env2}) \\
\qquad\qquad (\forall s\ .\quad a\ s \rightarrow T\ \mathit{env2}\ s \rightarrow \mathit{Env}\ t\ s\ \mathit{env1} \\
\qquad\qquad\quad \rightarrow (b\ s,\quad T\ \mathit{env1}\ s,\quad \mathit{Env}\ t\ s\ \mathit{env2}) \\
\qquad\qquad)
\end{array}
$$

Alternative: move the final s inwards

$$
\begin{array}{l}
\textbf{data}\ \mathit{Trafo2}\ m\ t\ a\ b = \\
\quad \mathit{Trafo2}\ (\forall \mathit{env1}\ .\ m\ \mathit{env1} \rightarrow \mathit{TrafoE2}\ m\ t\ a\ b\ \mathit{env1}) \\
\textbf{data}\ \mathit{TrafoE2}\ m\ t\ a\ b\ \mathit{env1} = \\
\quad \forall \mathit{env2}\ .\ \mathit{TrafoE2} \\
\qquad\qquad (m\ \mathit{env2}) \\
\qquad\qquad (\forall s\ .\ \ a\ s \rightarrow T\ \mathit{env2}\ s \rightarrow \mathit{Env}\ t\ s\ \mathit{env1} \\
\qquad\qquad\quad \rightarrow (b\ s,\ \ T\ \mathit{env1}\ s,\ \ \mathit{Env}\ t\ s\ \mathit{env2}) \\
\qquad\qquad)
\end{array}
$$

Unfortunately now a and b have an s parameter,and we can no longer use the arrow notation.

Conclusion

- The library was originally built for constructing the Left-Corner transform, which removes left-recursion from typed grammars (see our Haskell Workshop 2008 paper).

Conclusion

- The library was originally built for constructing the Left-Corner transform, which removes left-recursion from typed grammars (see our Haskell Workshop 2008 paper).
- The library has been used unmodified for left-factorisation of typed grammars, and the *cse* we have seen here.

Conclusion

- The library was originally built for constructing the Left-Corner transform, which removes left-recursion from typed grammars (see our Haskell Workshop 2008 paper).
- The library has been used unmodified for left-factorisation of typed grammars, and the *cse* we have seen here.
- Library is available from Hackage

Conclusion

- The library was originally built for constructing the Left-Corner transform, which removes left-recursion from typed grammars (see our Haskell Workshop 2008 paper).
- The library has been used unmodified for left-factorisation of typed grammars, and the cse we have seen here.
- Library is available from Hackage
- The library enables a whole new way of dealing with embedded domain specific languages.