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

January 24, 2009

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

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

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?

- 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:

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

GADTs

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

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

As a consequence our "compilers" do not have to perform type checking; 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:

$$
\begin{array}{l}
\textbf{data}\ Expr\ a\ env\ \textbf{where} \\
\qquad Var \quad :: Ref\ a\ env \rightarrow Expr\ a\ env \\
\qquad IntVal \ :: Int \qquad\quad \rightarrow Expr\ Int\ env \\
\qquad BoolVal :: Bool \qquad \rightarrow Expr\ Bool\ env \\
\qquad \ldots
\end{array}
$$

EDSL's may contain references

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

$$
\begin{array}{l}
\textbf{data}\ Expr\ a\ env\ \textbf{where} \\
\quad Var \quad\quad :: Ref\ a\ env \rightarrow Expr\ a\ env \\
\quad IntVal \quad :: Int \quad\quad\quad\quad \rightarrow Expr\ Int\ env \\
\quad BoolVal :: Bool \quad\quad\quad\quad \rightarrow Expr\ Bool\ env \\
\quad \ldots
\end{array}
$$

$$
\begin{array}{lll}
lookup :: Ref\ \ a\ env & \rightarrow env \rightarrow a & \\
eval \quad :: Expr\ a\ env & \rightarrow env \rightarrow a & \\
eval\ (Var\ r) \quad\quad e & = lookup\ r\ e & \\
eval\ (IntVal\ i) \quad \_ & = i & \\
eval\ (BoolVal\ b)\ \_ & = b & \\
eval\ (Add\ x\ y) \quad e & = eval\ x\ e + eval\ y\ e & \\
\ldots & &
\end{array}
$$

Sidestepping: Type equality

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

$$\begin{array}{l} \textbf{data}\ Equal :: \ * \ \rightarrow \ * \ \rightarrow \ * \ \textbf{where} \\ \quad Eq :: Equal\ a\ a \end{array}$$

Sidestepping: Type equality

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

$$\begin{array}{l} \textbf{data}\ Equal :: \ * \ \rightarrow \ * \ \rightarrow \ * \ \textbf{where} \\ \quad Eq :: Equal\ a\ a \end{array}$$

If a non-$\perp$ value $Eq\ a\ b$ 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:

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

Typed References

Environments are nested products

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

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

Typed References

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

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

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

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

Typed References

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

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

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

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

Here a and b are equal

Mutally Recursive Declarative Structures

We want to represent:

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

A first attempt:

$$\begin{aligned}\textbf{type}\ TwoLists = ((), &\ Expr\ [Int]\ TwoLists) \\ , &\ Expr\ [Int]\ TwoLists)\end{aligned}$$

Mutally Recursive Declarative Structures

We want to represent:

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

A first attempt:

$$\begin{aligned}\textbf{type}\ TwoLists = (((), &\ Expr\ [Int]\ TwoLists) \\ , &\ Expr\ [Int]\ TwoLists)\end{aligned}$$

Unfortunately this is not correct Haskell.

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:

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

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:

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

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:

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

Example

The expression:

We have nicer syntax for this

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

is now encoded as:

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

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:

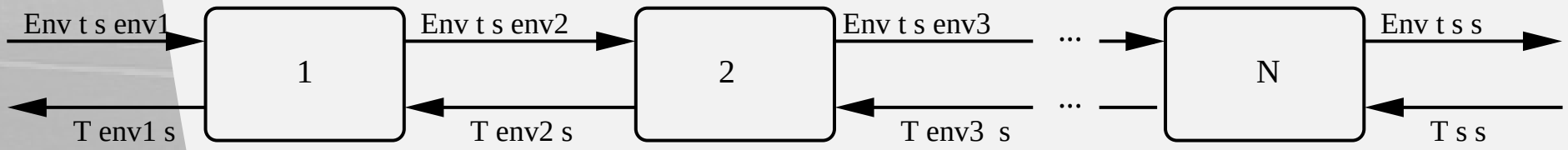
$$\mathbf{type}\ \mathit{FinalEnv}\ t\ \mathit{usedef} = \mathit{Env}\ t\ \mathit{usedef}\ \mathit{usedef}$$
$$\mathbf{newtype}\ T\ e\ s = T\{\mathit{unT} :: \forall x\ .\ \mathit{Ref}\ x\ e \rightarrow \mathit{Ref}\ x\ s\}$$

The Transformation Library

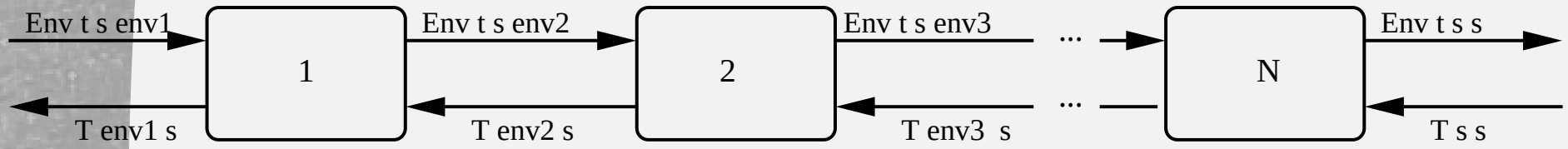
Eventually all references have to point into the final environment. We thus introduce the following types:

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

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



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

$\mathit{Env}\ t\ s\ \mathit{env1}$ the environment constructed thus far

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

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

$T\ \mathit{env1}\ s$ the updated $T\ \mathit{env2}\ s$

Extend with an arrow like interface

We extend the type with an arrow-like interface:

$$
\begin{array}{l}
\textbf{type}\ \mathit{Trafo}\ t\ s\ a\ b = \\
\quad \forall \mathit{env1}\ .\ \exists \mathit{env2}\ . \qquad a \rightarrow T\ \mathit{env2}\ s \rightarrow \mathit{Env}\ t\ s\ \mathit{env1} \\
\qquad\qquad\qquad\qquad \rightarrow (b, \quad T\ \mathit{env1}\ s, \quad \mathit{Env}\ t\ s\ \mathit{env2})
\end{array}
$$

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:

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

Haskelize

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

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

Creating a new Ref-erence

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

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

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

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

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

Creating a new *Ref*-erence

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

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

Tell the predecessors that an element was added

Creating a new *Ref*-erence

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

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

Map the newly added element into the final environment

Tell the predecessors that an element was added

Creating a new Ref-erence

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

Extend the environment with the new term ta

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

Map the newly added element into the final environment

Tell the predecessors that an element was added

Running the *Trafos*

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

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

Running the *Trafo*s

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

```
data Result m t b
    = ∀env2 . Result (m env2) (b env2)
                     (FinalEnv t 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
```

Running the *Trafo*s

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

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

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

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:

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

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

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

Common Subexpression Elimination

After CSE we have a larger, closed environment:

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

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

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

The type of cse now becomes:

$$cse :: Decls\ env \rightarrow TDecls\ env$$

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:

$$
\begin{array}{l}
\textbf{newtype}\ Memo\ env\ env' \\
\quad = Memo \\
\qquad (\forall x\ .\ Expr\ x\ env \\
\qquad\qquad \rightarrow Maybe\ (Ref\ x\ env') \\
\qquad)
\end{array}
$$

$$
\begin{array}{l}
emptyMemo :: Memo\ env\ () \\
emptyMemo = Memo\ (const\ Nothing)
\end{array}
$$

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:

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

Dealing with a single expression

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

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

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

Position of this expression

Where do the old variables go?

Dealing with a single expression

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

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

Running the transformations

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

Running the transformations

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

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

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

Running the transformations

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

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

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

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

$$
\begin{aligned}
&\mathit{cse} :: \forall \mathit{env}\ .\ \mathit{Decls}\ \mathit{env} \rightarrow \mathit{TDecls}\ \mathit{env} \\
&\mathit{cse}\ \mathit{decls} \\
&\quad = \mathbf{case}\ \mathit{runTrafo}\ (\mathit{trafo}\ \mathit{decls})\ \mathit{emptyMemo}\ ()\ \mathbf{of} \\
&\qquad\qquad \mathit{Result}\ \_\ t\ \mathit{env} \rightarrow \mathit{TDecls}\ \mathit{env}\ t
\end{aligned}
$$

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*

```
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)
```

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

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

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 librray 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 librray 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 librray 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.