Universiteit Utrecht

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

Talen en Compilers

2009/2010, periode 2

Andres Löh

Department of Information and Computing Sciences
Utrecht University

December 2, 2009

7. Compositional interpreters for expressions

This lecture

Compositional interpreters for expressions

- Reminder: simple expressions
- Variables
- Definitions
- Mutually recursive datatypes: declarations and expressions
- Using a list of declarations
- Use before definition

7.1 Reminder: simple expressions

Simple expressions

```
data E = Add E E
       | Neg E
       | Num Int

type EAlgebra r = (r → r → r,   -- add
                   r → r,       -- neg
                   Int → r)     -- num

foldE :: EAlgebra r → E → r
foldE (add, neg, num) = f
  where f (Add e1 e2) = add (f e1) (f e2)
        f (Neg e)     = neg (f e)
        f (Num n)     = num n
```

Evaluation

Directly:

$$
\begin{aligned}
&\mathsf{eval} :: \mathsf{E} \to \mathsf{Int} \\
&\mathsf{eval}\ (\mathsf{Add}\ \mathsf{e}_1\ \mathsf{e}_2) = \mathsf{eval}\ \mathsf{e}_1 + \mathsf{eval}\ \mathsf{e}_2 \\
&\mathsf{eval}\ (\mathsf{Neg}\ \mathsf{e}) \qquad\ \ = \mathsf{negate}\ (\mathsf{eval}\ \mathsf{e}) \\
&\mathsf{eval}\ (\mathsf{Num}\ \mathsf{n}) \qquad = \mathsf{n}
\end{aligned}
$$

Evaluation

Directly:

$$
\begin{aligned}
&\mathsf{eval} :: \mathsf{E} \to \mathsf{Int} \\
&\mathsf{eval}\ (\mathsf{Add}\ \mathsf{e}_1\ \mathsf{e}_2) = \mathsf{eval}\ \mathsf{e}_1 + \mathsf{eval}\ \mathsf{e}_2 \\
&\mathsf{eval}\ (\mathsf{Neg}\ \mathsf{e}) \qquad\quad = \mathsf{negate}\ (\mathsf{eval}\ \mathsf{e}) \\
&\mathsf{eval}\ (\mathsf{Num}\ \mathsf{n}) \qquad\ \ = \mathsf{n}
\end{aligned}
$$

Using foldE:

$$
\begin{aligned}
&\mathsf{evalAlgebra} :: \mathsf{EAlgebra}\ \mathsf{Int} \\
&\mathsf{evalAlgebra} = ((+), \mathsf{negate}, \mathsf{id}) \\
&\mathsf{eval} :: \mathsf{E} \to \mathsf{Int} \\
&\mathsf{eval} = \mathsf{foldE}\ \mathsf{evalAlgebra}
\end{aligned}
$$

7.2 Variables

Adding variables

Let us consider expressions with variables:

```
data E = Add E E
       | Neg E
       | Num Int
```

Adding variables

Let us consider expressions with variables:

```
data E = Add E E
       | Neg E
       | Num Int
       | Var Id
```

Adding variables

Let us consider expressions with variables:

```
data E = Add E E
       | Neg E
       | Num Int
       | Var Id
```

We use strings to represent identifiers:

```
type Id = String
```

Extending algebra and fold

```
type EAlgebra r = (r → r → r,    -- add
                   r → r,         -- neg
                   Int → r,       -- num
                   Id → r)        -- var
```

Extending algebra and fold

```
type EAlgebra r = (r → r → r,   -- add
                   r → r,       -- neg
                   Int → r,     -- num
                   Id → r)      -- var
```

```
foldE :: EAlgebra r → E → r
foldE (add, neg, num, var) = f
  where f (Add e1 e2) = add (f e1) (f e2)
        f (Neg e)     = neg (f e)
        f (Num n)     = num n
        f (Var x)     = var x
```

Evaluating expressions with variables

Question

What is the value of the following expression?

```
- x + 1
```

Evaluating expressions with variables

Question

What is the value of the following expression?

```
- x + 1
```

Observation

We have to know the (integer) value of x if we want to assign an (integer) value to the expression.

Free variables

Similarly, in order to assign an integer value to

```
x + y + y + z
```

we have to know the integer values of x, y and z.

Free variables

Similarly, in order to assign an integer value to

```
x + y + y + z
```

we have to know the integer values of x, y and z.

Variables in an expression that are not defined within the expression itself are called **free variables**.

Free variables

Similarly, in order to assign an integer value to

```
x + y + y + z
```

we have to know the integer values of x, y and z.

Variables in an expression that are not defined within the expression itself are called **free variables**.

In order to determine the value of an expression, we have to know the values of the free variables that occur in the expression.

Evaluating expressions with variables

In other words, the value of an expression possibly containing free variables is not an Int, but a function

> Env $\rightarrow$ Int

where Env is an **environment** mapping the free variables to integer values.

Representing an environment

We need a mapping from identifiers (type Id) to values (here type Int). There are several ways to implement such a mapping:

Representing an environment

We need a mapping from identifiers (type Id) to values (here type Int). There are several ways to implement such a mapping:

Lists of pairs

```
type Env = [(Id, Int)]
```

Insert in $O(1)$, lookup in $O(n)$.

Representing an environment

We need a mapping from identifiers (type Id) to values (here type Int). There are several ways to implement such a mapping:

Lists of pairs

```
type Env = [(Id, Int)]
```

Insert in $O(1)$, lookup in $O(n)$.

Finite maps

```
import Data.Map
type Env = Map Id Int
```

Implemented using balanced trees. Insert/lookup in $O(\log n)$.

Interface of finite maps

```
import Data.Map as M   -- short name to disambiguate
type Map k v   abstract
M.empty :: Map k v   -- not the parser combinator
insert  :: Ord k ⇒ k → v → Map k v → Map k v
(!)     :: Ord k ⇒ Map k v → k → v
```

Question

Why the Ord constraint?

Evaluation with variables

Directly:

$$
\begin{array}{ll}
\mathsf{eval} :: \mathsf{E} \rightarrow \mathsf{Env} \rightarrow \mathsf{Int} & \\
\mathsf{eval}\ (\mathsf{Add}\ e_1\ e_2) & \mathsf{env} = \mathsf{eval}\ e_1\ \mathsf{env} + \mathsf{eval}\ e_2\ \mathsf{env} \\
\mathsf{eval}\ (\mathsf{Neg}\ e) & \mathsf{env} = \mathsf{negate}\ (\mathsf{eval}\ e\ \mathsf{env}) \\
\mathsf{eval}\ (\mathsf{Num}\ n) & \mathsf{env} = n \\
\mathsf{eval}\ (\mathsf{Var}\ x) & \mathsf{env} = \mathsf{env}\ !\ x
\end{array}
$$

Evaluation with variables

Directly:

$$
\begin{array}{ll}
\mathsf{eval} :: \mathsf{E} \to \mathsf{Env} \to \mathsf{Int} & \\
\mathsf{eval}\ (\mathsf{Add}\ \mathsf{e}_1\ \mathsf{e}_2) & \mathsf{env} = \mathsf{eval}\ \mathsf{e}_1\ \mathsf{env} + \mathsf{eval}\ \mathsf{e}_2\ \mathsf{env} \\
\mathsf{eval}\ (\mathsf{Neg}\ \mathsf{e}) & \mathsf{env} = \mathsf{negate}\ (\mathsf{eval}\ \mathsf{e}\ \mathsf{env}) \\
\mathsf{eval}\ (\mathsf{Num}\ \mathsf{n}) & \mathsf{env} = \mathsf{n} \\
\mathsf{eval}\ (\mathsf{Var}\ \mathsf{x}) & \mathsf{env} = \mathsf{env}\ !\ \mathsf{x}
\end{array}
$$

Algebra:

$$
\begin{array}{l}
\mathsf{evalAlgebra} :: \mathsf{EAlgebra}\ (\mathsf{Env} \to \mathsf{Int}) \\
\mathsf{evalAlgebra} = \\
\quad \begin{array}{lll}
(\lambda \mathsf{r}_1\ \mathsf{r}_2 & \to \lambda \mathsf{env} \to & \mathsf{r}_1\ \mathsf{env} + \mathsf{r}_2\ \mathsf{env}, \\
\ \lambda \mathsf{r} & \to \lambda \mathsf{env} \to & \mathsf{negate}\ (\mathsf{r}\ \mathsf{env}), \\
\ \lambda \mathsf{n} & \to \lambda \mathsf{env} \to & \mathsf{n}, \\
\ \lambda \mathsf{x} & \to \lambda \mathsf{env} \to & \mathsf{env}\ !\ \mathsf{x})
\end{array}
\end{array}
$$

Where to place the environment?

Whats the difference between the following two algebras?

$$
\begin{array}{l}
\mathsf{evalAlgebra} :: \mathsf{EAlgebra}\ (\mathsf{Env} \to \mathsf{Int}) \\
\mathsf{evalAlgebra} = \\
\quad (\lambda r_1\ r_2 \to \lambda env \to r_1\ env + r_2\ env, \\
\quad\ \ \lambda r \quad\ \ \to \lambda env \to negate\ (r\ env), \\
\quad\ \ \lambda n \quad\ \ \to \lambda env \to n, \\
\quad\ \ \lambda x \quad\ \ \to \lambda env \to env\ !\ x)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{evalAlgebra} :: \mathsf{Env} \to \mathsf{EAlgebra\ Int} \\
\mathsf{evalAlgebra}\ env = \\
\quad (\lambda r_1\ r_2 \to r_1 + r_2, \\
\quad\ \ \lambda r \quad\ \ \to negate\ r, \\
\quad\ \ \lambda n \quad\ \ \to n, \\
\quad\ \ \lambda x \quad\ \ \to env\ !\ x)
\end{array}
$$

7.3 Definitions

Adding definitions

Passing the environment within the algebra is useful if we want to change the environment during evaluation – power we do not yet need, but that is helpful as soon as we add (local) definitions to the language.

Adding definitions

Passing the environment within the algebra is useful if we want to change the environment during evaluation – power we do not yet need, but that is helpful as soon as we add (local) definitions to the language.

```
data E = Add E E
       | Neg E
       | Num Int
       | Var Id
       | Def Id E E   -- new: let x = e1 in e2
```

Extending algebra and fold

```
type EAlgebra r = (r → r → r,          -- add
                   r → r,              -- neg
                   Int → r,            -- num
                   Id → r,             -- var
                   Id → r → r → r)     -- def
```

Extending algebra and fold

```
type EAlgebra r = (r → r → r,          -- add
                   r → r,              -- neg
                   Int → r,            -- num
                   Id → r,             -- var
                   Id → r → r → r)     -- def
```

```
foldE :: EAlgebra r → E → r
foldE (add, neg, num, var) = f
  where f (Add e1 e2)   = add (f e1) (f e2)
        f (Neg e)       = neg (f e)
        f (Num n)       = num n
        f (Var x)       = var x
        f (Def x e1 e2) = def x (f e1) (f e2)
```

Considerations for defining evaluation

What should the following expressions evaluate to?

```
let x = 1 in x
let x = y in x + x
let x = 1 in let x = 2 in x
let x = 1 in let x = x + 1 in x
```

Considerations for defining evaluation

What should the following expressions evaluate to?

```
let x = 1 in x
let x = y in x + x
let x = 1 in let x = 2 in x
let x = 1 in let x = x + 1 in x
```

We observe and decide:

- in general, we still need an environment, even if we can now define **closed terms** with variables;
- inner definitions should shadow outer definitions;
- since we cannot make useful definitions using recursion, we do not make the bound variable available on the right hand side of the binding.

Evaluating expressions with definitions

Directly:

$$\begin{aligned}
&\mathsf{eval} :: \mathsf{E} \to \mathsf{Env} \to \mathsf{Int} \\
&\ldots \quad \text{-- as before} \\
&\mathsf{eval}\ (\mathsf{Def}\ x\ e_1\ e_2)\ \mathsf{env} = \mathsf{eval}\ e_2\ (\mathsf{insert}\ x\ (\mathsf{eval}\ e_1\ \mathsf{env})\ \mathsf{env})
\end{aligned}$$

- Evaluate e_1 in the outer environment env.
- Value is bound to x and inserted into the environment env.
- Evaluate e_2 in the resulting environment.

Evaluating expressions with definitions

Directly:

$$\begin{aligned}&\mathsf{eval} :: \mathsf{E} \to \mathsf{Env} \to \mathsf{Int}\\ &\ldots \quad \text{-- as before}\\ &\mathsf{eval}\ (\mathsf{Def}\ \mathsf{x}\ \mathsf{e}_1\ \mathsf{e}_2)\ \mathsf{env} = \mathsf{eval}\ \mathsf{e}_2\ (\mathsf{insert}\ \mathsf{x}\ (\mathsf{eval}\ \mathsf{e}_1\ \mathsf{env})\ \mathsf{env})\end{aligned}$$

- Evaluate e_1 in the outer environment env.
- Value is bound to x and inserted into the environment env.
- Evaluate e_2 in the resulting environment.

Algebra:

$$\begin{aligned}&\mathsf{evalAlgebra} :: \mathsf{EAlgebra}\ (\mathsf{Env} \to \mathsf{Int})\\ &\mathsf{evalAlgebra} =\\ &\quad (\ldots, \quad \text{-- as before}\\ &\quad\ \lambda \mathsf{x}\ \mathsf{r}_1\ \mathsf{r}_2 \to \lambda \mathsf{env} \to \mathsf{r}_2\ (\mathsf{insert}\ \mathsf{x}\ (\mathsf{r}_1\ \mathsf{env})\ \mathsf{env}))\end{aligned}$$

7.4 Mutually recursive datatypes: declarations and expressions

Abstracting from declarations

```
data E = Add E E
       | Neg E
       | Num Int
       | Var Id
       | Def Id E E
```

Abstracting from declarations

$$
\begin{aligned}
\textbf{data}\ E = {} & \mathsf{Add}\ E\ E \\
\mid {} & \mathsf{Neg}\ E \\
\mid {} & \mathsf{Num}\ \mathsf{Int} \\
\mid {} & \mathsf{Var}\ \mathsf{Id} \\
\mid {} & \mathsf{Def}\ \boxed{\mathsf{Id}\ E}\ E
\end{aligned}
$$

Abstracting from declarations

```
data E = Add E E
       | Neg E
       | Num Int
       | Var Id
       | Def D E

data D = Dcl Id E
```

Abstracting from declarations

```
data E = Add E E
       | Neg E
       | Num Int
       | Var Id
       | Def D E

data D = Dcl Id E
```

How does this change affect the algebra and fold?

Algebra for families of datatypes

Each datatype in the family can have its own result type.

$$
\begin{aligned}
\mathsf{Add} &:: \mathsf{E} \to \mathsf{E} \to \mathsf{E}\\
\mathsf{Neg} &:: \mathsf{E} \to \mathsf{E}\\
\mathsf{Num} &:: \mathsf{Int} \to \mathsf{E}\\
\mathsf{Var} &:: \mathsf{Id} \to \mathsf{E}\\
\mathsf{Def} &:: \mathsf{D} \to \mathsf{E} \to \mathsf{E}\\
\mathsf{Dcl} &:: \mathsf{Id} \to \mathsf{E} \to \mathsf{D}
\end{aligned}
$$

Result type e for expressions, result type d for declarations:

$$
\begin{aligned}
&\textbf{type}\ \mathsf{EDAlgebra}\ \mathsf{e}\ \mathsf{d} =\\
&\quad (\mathsf{e} \to \mathsf{e} \to \mathsf{e},\\
&\quad\ \ \mathsf{e} \to \mathsf{e},\\
&\quad\ \ \mathsf{Int} \to \mathsf{e},\\
&\quad\ \ \mathsf{Id} \to \mathsf{e},\\
&\quad\ \ \mathsf{d} \to \mathsf{e} \to \mathsf{e},\\
&\quad\ \ \mathsf{Id} \to \mathsf{e} \to \mathsf{d})
\end{aligned}
$$

Universiteit Utrecht

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

Fold for families of datatypes

We also need one function per type to traverse the structure:

$$
\begin{array}{l}
\mathsf{foldE} :: \mathsf{EDAlgebra\ e\ d} \to \mathsf{Expr} \to \mathsf{e} \\
\mathsf{foldE}\ (\mathsf{add}, \mathsf{neg}, \mathsf{num}, \mathsf{var}, \mathsf{def}, \mathsf{dcl}) = \mathsf{fe} \\
\quad \textbf{where}\ \begin{array}[t]{ll}
\mathsf{fe}\ (\mathsf{Add}\ \mathsf{e}_1\ \mathsf{e}_2) & = \mathsf{add}\ (\mathsf{fe}\ \mathsf{e}_1)\ (\mathsf{fe}\ \mathsf{e}_2) \\
\mathsf{fe}\ (\mathsf{Neg}\ \mathsf{e}) & = \mathsf{neg}\ (\mathsf{fe}\ \mathsf{e}) \\
\mathsf{fe}\ (\mathsf{Num}\ \mathsf{n}) & = \mathsf{num}\ \mathsf{n} \\
\mathsf{fe}\ (\mathsf{Var}\ \mathsf{x}) & = \mathsf{var}\ \mathsf{x} \\
\mathsf{fe}\ (\mathsf{Def}\ \mathsf{d}\ \mathsf{e}) & = \mathsf{def}\ (\mathsf{fd}\ \mathsf{d})\ (\mathsf{fe}\ \mathsf{e}) \\
\mathsf{fd}\ (\mathsf{Dcl}\ \mathsf{x}\ \mathsf{e}) & = \mathsf{dcl}\ \mathsf{x}\ (\mathsf{fe}\ \mathsf{e})
\end{array}
\end{array}
$$

Fold for families of datatypes

We also need one function per type to traverse the structure:

$$
\begin{array}{l}
\mathsf{foldE} :: \mathsf{EDAlgebra\ e\ d} \rightarrow \mathsf{Expr} \rightarrow \mathsf{e} \\
\mathsf{foldE}\ (\mathsf{add}, \mathsf{neg}, \mathsf{num}, \mathsf{var}, \mathsf{def}, \mathsf{dcl}) = \mathsf{fe} \\
\quad \textbf{where}\ \begin{array}{lll}
\mathsf{fe}\ (\mathsf{Add}\ \mathsf{e}_1\ \mathsf{e}_2) & = & \mathsf{add}\ (\mathsf{fe}\ \mathsf{e}_1)\ (\mathsf{fe}\ \mathsf{e}_2) \\
\mathsf{fe}\ (\mathsf{Neg}\ \mathsf{e}) & = & \mathsf{neg}\ (\mathsf{fe}\ \mathsf{e}) \\
\mathsf{fe}\ (\mathsf{Num}\ \mathsf{n}) & = & \mathsf{num}\ \mathsf{n} \\
\mathsf{fe}\ (\mathsf{Var}\ \mathsf{x}) & = & \mathsf{var}\ \mathsf{x} \\
\mathsf{fe}\ (\mathsf{Def}\ \mathsf{d}\ \mathsf{e}) & = & \mathsf{def}\ (\boxed{\mathsf{fd}\ \mathsf{d}})\ (\mathsf{fe}\ \mathsf{e}) \\
\mathsf{fd}\ (\mathsf{Dcl}\ \mathsf{x}\ \mathsf{e}) & = & \mathsf{dcl}\ \mathsf{x}\ (\boxed{\mathsf{fe}\ \mathsf{e}})
\end{array}
\end{array}
$$

Adapting evaluation (directly)

Question

What is the best result type to choose for a declaration?

Adapting evaluation (directly)

Question

What is the best result type to choose for a declaration?

$$
\begin{array}{ll}
\mathsf{evalE} :: \mathsf{E} \rightarrow \mathsf{Env} \rightarrow \mathsf{Int} & \\
\mathsf{evalE}\ (\mathsf{Add}\ \mathsf{e}_1\ \mathsf{e}_2) & \mathsf{env} = \mathsf{evalE}\ \mathsf{e}_1\ \mathsf{env} + \mathsf{evalE}\ \mathsf{e}_2\ \mathsf{env} \\
\mathsf{evalE}\ (\mathsf{Num}\ \mathsf{e}) & \mathsf{env} = \mathsf{negate}\ (\mathsf{evalE}\ \mathsf{e}\ \mathsf{env}) \\
\mathsf{evalE}\ (\mathsf{Num}\ \mathsf{n}) & \mathsf{env} = \mathsf{n} \\
\mathsf{evalE}\ (\mathsf{Var}\ \mathsf{x}) & \mathsf{env} = \mathsf{env}\ !\ \mathsf{x} \\
\mathsf{evalE}\ (\mathsf{Def}\ \mathsf{d}\ \mathsf{e}) & \mathsf{env} = \mathsf{evalE}\ \mathsf{e}\ (\mathsf{evalD}\ \mathsf{d}\ \mathsf{env}) \\
\mathsf{evalD} :: \mathsf{D} \rightarrow \mathsf{Env} \rightarrow \mathsf{Env} & \\
\mathsf{evalD}\ (\mathsf{Dcl}\ \mathsf{x}\ \mathsf{e}) & \mathsf{env} = \mathsf{insert}\ \mathsf{x}\ (\mathsf{evalE}\ \mathsf{e}\ \mathsf{env})\ \mathsf{env}
\end{array}
$$

Adapting evaluation (as a fold)

$$
\begin{array}{l}
\mathsf{evalAlgebra} :: \mathsf{EDAlgebra}\ (\mathsf{Env} \to \mathsf{Int})\ (\mathsf{Env} \to \mathsf{Env}) \\
\mathsf{evalAlgebra} = \\
\quad \begin{array}{lll}
(\lambda \mathsf{e}_1\ \mathsf{e}_2 & \to \lambda \mathsf{env} & \to \mathsf{e}_1\ \mathsf{env} + \mathsf{e}_2\ \mathsf{env}, \\
\ \lambda \mathsf{e} & \to \lambda \mathsf{env} & \to \mathsf{negate}\ (\mathsf{e}\ \mathsf{env}), \\
\ \lambda \mathsf{n} & \to \lambda \mathsf{env} & \to \mathsf{n}, \\
\ \lambda \mathsf{x} & \to \lambda \mathsf{env} & \to \mathsf{env}\ !\ \mathsf{x}, \\
\ \lambda \mathsf{d}\ \mathsf{e} & \to \lambda \mathsf{env} & \to \mathsf{e}\ (\mathsf{d}\ \mathsf{env}), \\
\ \lambda \mathsf{x}\ \mathsf{e} & \to \lambda \mathsf{env} & \to \mathsf{insert}\ \mathsf{x}\ (\mathsf{e}\ \mathsf{env})\ \mathsf{env})
\end{array}
\end{array}
$$

7.5 Using a list of declarations

Multiple declarations per definition

```
data E = Add E E
       | Neg E
       | Num Int
       | Var Id
       | Def [D] E    -- modified
data D = Dcl Id E
```

Multiple declarations per definition

```
data E = Add E E
       | Neg E
       | Num Int
       | Var Id
       | Def [D] E    -- modified
data D = Dcl Id E
```

We could also have created a new datatype:

```
data E  = ...
        | Def Ds E
data Ds = NoD
        | OneD Ds D
```

A **snoc-list** seems slightly advantageous – why?

Adapting the algebra and fold

We keep the list in the algebra ...

```
type EDAlgebra e d =
  ( ...,
   [d] → e → e,
   ... )
```

Adapting the algebra and fold

We keep the list in the algebra ...

```
type EDAlgebra e d =
   ( ...,
    [d] → e → e,
    ... )
```

... and use map in the fold function:

```
foldE :: EDAlgebra e d → Expr → e
foldE (add, neg, num, var, def, dcl) = fe
   where ...
         fe (Def ds e) = def (map fd ds) (fe e)
         ...
```

Adapting evaluation

We now get a list of Env $\rightarrow$ Env functions (one for each declaration) in the case for Def:

Adapting evaluation

We now get a list of Env → Env functions (one for each declaration) in the case for Def:

$$
\begin{array}{l}
\mathsf{evalAlgebra} :: \mathsf{EDAlgebra}\ (\mathsf{Env} \to \mathsf{Int})\ (\mathsf{Env} \to \mathsf{Env}) \\
\mathsf{evalAlgebra} = \\
\quad (\lambda \mathsf{e}_1\ \mathsf{e}_2 \to \lambda \mathsf{env} \to \mathsf{e}_1\ \mathsf{env} + \mathsf{e}_2\ \mathsf{env}, \\
\quad\ \ \lambda \mathsf{e} \quad\ \ \to \lambda \mathsf{env} \to \mathsf{negate}\ (\mathsf{e}\ \mathsf{env}), \\
\quad\ \ \lambda \mathsf{n} \quad\ \ \to \lambda \mathsf{env} \to \mathsf{n}, \\
\quad\ \ \lambda \mathsf{x} \quad\ \ \to \lambda \mathsf{env} \to \mathsf{env}\ !\ \mathsf{x}, \\
\quad\ \ \lambda \mathsf{ds}\ \mathsf{e} \to \lambda \mathsf{env} \to \mathsf{e}\ (\mathsf{process}\ \mathsf{ds}\ \mathsf{env}), \\
\quad\ \ \lambda \mathsf{x}\ \mathsf{e} \ \ \to \lambda \mathsf{env} \to \mathsf{insert}\ \mathsf{x}\ (\mathsf{e}\ \mathsf{env})\ \mathsf{env})
\end{array}
$$

Adapting evaluation

We now get a list of Env → Env functions (one for each declaration) in the case for Def:

$$
\begin{array}{l}
\mathsf{evalAlgebra} :: \mathsf{EDAlgebra}\ (\mathsf{Env} \to \mathsf{Int})\ (\mathsf{Env} \to \mathsf{Env}) \\
\mathsf{evalAlgebra} = \\
\quad \begin{array}{lll}
(\lambda \mathsf{e}_1\ \mathsf{e}_2 & \to \lambda \mathsf{env} & \to \mathsf{e}_1\ \mathsf{env} + \mathsf{e}_2\ \mathsf{env}, \\
\ \lambda \mathsf{e} & \to \lambda \mathsf{env} & \to \mathsf{negate}\ (\mathsf{e}\ \mathsf{env}), \\
\ \lambda \mathsf{n} & \to \lambda \mathsf{env} & \to \mathsf{n}, \\
\ \lambda \mathsf{x} & \to \lambda \mathsf{env} & \to \mathsf{env}\ !\ \mathsf{x}, \\
\ \lambda \mathsf{ds}\ \mathsf{e} & \to \lambda \mathsf{env} & \to \mathsf{e}\ (\mathsf{process}\ \mathsf{ds}\ \mathsf{env}), \\
\ \lambda \mathsf{x}\ \mathsf{e} & \to \lambda \mathsf{env} & \to \mathsf{insert}\ \mathsf{x}\ (\mathsf{e}\ \mathsf{env})\ \mathsf{env})
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{process} :: [\mathsf{Env} \to \mathsf{Env}] \to \mathsf{Env} \to \mathsf{Env} \\
\mathsf{process}\ \mathsf{ds}\ \mathsf{env} = \mathsf{foldl}\ (\mathsf{flip}\ (\$))\ \mathsf{env}\ \mathsf{ds}
\end{array}
$$

7.6 Use before definition

Recursion revisited

We said before that we interpret

```
let x = x + 1 in ...
```

as a redefinition because we have no meaningful way of defining recursive functions yet.

Recursion revisited

We said before that we interpret

```
let x = x + 1 in ...
```

as a redefinition because we have no meaningful way of defining recursive functions yet.

Let us now reconsider this decision and assume that we want

```
let x = y + 1;
    y = 2;
    z = x + y + 3
in  z
```

to be allowed and evaluate to 8.

Evaluating recursive declarations

The result type for declarations now becomes

> $\mathsf{Env} \to \mathsf{Env} \to \mathsf{Env}$

We pass **two** environments:

- the **current** environment about to be extended,
- the **final** environment that already is extended (boldly assuming that we already know that).

Evaluating recursive declarations

The result type for declarations now becomes

> Env → Env → Env

We pass **two** environments:

- the **current** environment about to be extended,
- the **final** environment that already is extended (boldly assuming that we already know that).

We use the final environment to evaluate the right hand sides.

We extend the current environment one by one.

Evaluating recursive declarations

The result type for declarations now becomes

> Env → Env → Env

We pass **two** environments:

- the **current** environment about to be extended,
- the **final** environment that already is extended (boldly assuming that we already know that).

We use the final environment to evaluate the right hand sides.

We extend the current environment one by one.

In the end, we tie the knot as follows:

> **let** finalenv = process ds currentenv finalenv **in** . . .

Adapting the fold

$$
\begin{array}{l}
\mathsf{evalAlgebra} :: \mathsf{EDAlgebra}\ (\mathsf{Env} \to \mathsf{Int})\ (\mathsf{Env} \to \mathsf{Env} \to \mathsf{Env}) \\
\mathsf{evalAlgebra} = \\
\quad \begin{array}{llll}
(\lambda \mathsf{e}_1\ \mathsf{e}_2 & \to \lambda \mathsf{env} & \to & \mathsf{e}_1\ \mathsf{env} + \mathsf{e}_2\ \mathsf{env}, \\
\ \lambda \mathsf{e} & \to \lambda \mathsf{env} & \to & \mathsf{negate}\ (\mathsf{e}\ \mathsf{env}), \\
\ \lambda \mathsf{n} & \to \lambda \mathsf{env} & \to & \mathsf{n}, \\
\ \lambda \mathsf{x} & \to \lambda \mathsf{env} & \to & \mathsf{env}\ !\ \mathsf{x}, \\
\ \lambda \mathsf{ds}\ \mathsf{e} & \to \lambda \mathsf{env} & \to & \mathbf{let}\ \mathsf{fenv} = \mathsf{process}\ \mathsf{ds}\ \mathsf{env}\ \mathsf{fenv} \\
 & & & \mathbf{in}\ \mathsf{e}\ \mathsf{fenv}, \\
\ \lambda \mathsf{x}\ \mathsf{e} & \to \lambda \mathsf{env}\ \mathsf{fenv} & \to & \mathsf{insert}\ \mathsf{x}\ (\mathsf{e}\ \mathsf{fenv})\ \mathsf{env})
\end{array}
\end{array}
$$

Adapting the fold

$$
\begin{aligned}
&\mathsf{evalAlgebra} :: \mathsf{EDAlgebra}\ (\mathsf{Env} \to \mathsf{Int})\ (\mathsf{Env} \to \mathsf{Env} \to \mathsf{Env})\\
&\mathsf{evalAlgebra} =\\
&\quad (\lambda e_1\ e_2 \to \lambda \mathsf{env} \to e_1\ \mathsf{env} + e_2\ \mathsf{env},\\
&\quad\ \lambda e \to \lambda \mathsf{env} \to \mathsf{negate}\ (e\ \mathsf{env}),\\
&\quad\ \lambda n \to \lambda \mathsf{env} \to n,\\
&\quad\ \lambda x \to \lambda \mathsf{env} \to \mathsf{env}\ !\ x,\\
&\quad\ \lambda ds\ e \to \lambda \mathsf{env} \to \mathbf{let}\ \mathsf{fenv} = \mathsf{process}\ ds\ \mathsf{env}\ \mathsf{fenv}\\
&\quad\qquad\qquad\qquad\qquad\ \mathbf{in}\ e\ \mathsf{fenv},\\
&\quad\ \lambda x\ e \to \lambda \mathsf{env}\ \mathsf{fenv} \to \mathsf{insert}\ x\ (e\ \mathsf{fenv})\ \mathsf{env})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{process} :: [\mathsf{Env} \to \mathsf{Env} \to \mathsf{Env}] \to \mathsf{Env} \to \mathsf{Env} \to \mathsf{Env}\\
&\mathsf{process}\ ds\ \mathsf{env}\ \mathsf{fenv} = \mathsf{foldl}\ (\lambda \mathsf{cenv}\ d \to d\ \mathsf{cenv}\ \mathsf{fenv})\ \mathsf{env}\ ds
\end{aligned}
$$

Summary

- We can define algebras and folds also for families of mutually recursive types, and also if lists (or other types) occur surrounding the recursive positions.
- Often, the result types of algebras are themselves functions.
- Function arguments represent information that is distributed over the abstract syntax tree.
- Function results represent information that is computed from the abstract syntax tree (and the distributed values).
- Next lecture: summary and the bigger picture.