Universiteit Utrecht

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

Talen en Compilers

2023 - 2024

David van Balen

Department of Information and Computing Sciences
Utrecht University

2023-12-12

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:

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

Evaluation

Directly:

$$\begin{aligned} &\text{eval} :: \text{E} \rightarrow \text{Int} \\ &\text{eval (Add } e_1\ e_2) = \text{eval } e_1 + \text{eval } e_2 \\ &\text{eval (Neg e)} \quad\;\; = \text{negate (eval e)} \\ &\text{eval (Num n)} \quad = \text{n} \end{aligned}$$

Using foldE:

$$\begin{aligned} &\text{eval} :: \text{E} \rightarrow \text{Int} \\ &\text{eval} = \text{foldE } ((+), \text{negate}, \text{id}) \end{aligned}$$

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

What is the value of the following expression?

```
-x + 1
```

Evaluating expressions with variables

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 (dictionaries)

```
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 (Map)
import qualfified Data.Map as Map

type Map k v   -- abstract

Map.empty :: Map k v
Map.insert :: Ord k ⇒ k → v → Map k v → Map k v
(Map.!)     :: Ord k ⇒ Map k v → k → v
```

Evaluation with variables

Directly:

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

Evaluation with variables

Directly:

```
eval :: E → Env → Int
eval (Add e₁ e₂) env = eval e₁ env + eval e₂ env
eval (Neg e)     env = negate (eval e env)
eval (Num n)     env = n
eval (Var x)     env = env ! x
```

Algebra:

```
evalAlgebra :: EAlgebra (Env → Int)
evalAlgebra =
  (λr₁ r₂ → λenv → r₁ env + r₂ env,
   λr     → λenv → negate (r env),
   λn     → λenv → n,
   λx     → λenv → env ! x)
```

Where to place the environment?

What is the difference between the following two algebras?

```
evalAlgebra :: EAlgebra (Env → Int)
evalAlgebra =
   (λr₁ r₂ → λenv → r₁ env + r₂ env,
    λr     → λenv → negate (r env),
    λn     → λenv → n,
    λx     → λenv → env ! x)
evalAlgebra :: Env → EAlgebra Int
evalAlgebra env =
   (λr₁ r₂ → r₁ + r₂,
    λr     → negate r,
    λn     → n,
    λx     → env ! x)
```

7.3 Definitions

Adding definitions

$\textbf{let}\ x = 3\ \textbf{in}\ x + 1$

$x \leftarrow 3;$
$x + 1$

Adding definitions

let $\mathsf{x} = 3$ **in** $\mathsf{x} + 1$

$\mathsf{x} \leftarrow 3;$
$\mathsf{x} + 1$

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

Adding definitions

$\textbf{let}\ x = 3\ \textbf{in}\ x + 1$

$x \leftarrow 3;$
$x + 1$

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

```
Def "x" (Num 3) (Add (Var "x") (Num 1))
```

Extending algebra and fold

```
data EAlgebra r = EAlg {
  {add :: r → r → r,
  , neg ::  r → r,
  , num :: Int → r,
  , var ::  Id → r,
  , def ::  Id → r → r → r}
```

Extending algebra and fold

```
data EAlgebra r = EAlg {
   {add :: r → r → r,
   , neg ::  r → r,
   , num :: Int → r,
   , var ::  Id → r,
   , def ::  Id → r → r → r}
```

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

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

- 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:

```
eval :: E → Env → Int
...    -- as before
eval (Def x e1 e2) env = eval e2 (insert x (eval e1 env) env)
```

- 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:

```
evalAlgebra :: EAlgebra (Env → Int)
evalAlgebra = EAlg-- as before
```


Universiteit Utrecht

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

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

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

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?

Universiteit Utrecht

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

Algebra for families of datatypes

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

```
Add :: E → E → E
Neg :: E → E
Num :: Int → E
Var :: Id → E
Def :: D → E → E

Dcl :: Id → E → D
```

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

```
type EDAlgebra e d =
  (e → e → e,
   e → e,
   Int → e,
   Id → e,
   d → e → e,

   Id → e → d)
```

Fold for families of datatypes

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

$$
\begin{aligned}
&\mathsf{foldE} :: \mathsf{EDAlgebra\ e\ d} \to \mathsf{E} \to \mathsf{e}\\
&\mathsf{foldE}\ (\mathsf{add}, \mathsf{neg}, \mathsf{num}, \mathsf{var}, \mathsf{def}, \mathsf{dcl}) = \mathsf{fe}\\
&\quad \textbf{where}\ \mathsf{fe}\ (\mathsf{Add}\ \mathsf{e}_1\ \mathsf{e}_2) = \mathsf{add}\ (\mathsf{fe}\ \mathsf{e}_1)\ (\mathsf{fe}\ \mathsf{e}_2)\\
&\qquad\qquad \mathsf{fe}\ (\mathsf{Neg}\ \mathsf{e}) = \mathsf{neg}\ (\mathsf{fe}\ \mathsf{e})\\
&\qquad\qquad \mathsf{fe}\ (\mathsf{Num}\ \mathsf{n}) = \mathsf{num}\ \mathsf{n}\\
&\qquad\qquad \mathsf{fe}\ (\mathsf{Var}\ \mathsf{x}) = \mathsf{var}\ \mathsf{x}\\
&\qquad\qquad \mathsf{fe}\ (\mathsf{Def}\ \mathsf{d}\ \mathsf{e}) = \mathsf{def}\ (\mathsf{fd}\ \mathsf{d})\ (\mathsf{fe}\ \mathsf{e})\\
&\qquad\qquad \mathsf{fd}\ (\mathsf{Dcl}\ \mathsf{x}\ \mathsf{e}) = \mathsf{dcl}\ \mathsf{x}\ (\mathsf{fe}\ \mathsf{e})
\end{aligned}
$$

Fold for families of datatypes

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

```
foldE :: EDAlgebra e d → E → e
foldE (add, neg, num, var, def, dcl) = fe
  where fe (Add e1 e2) = add (fe e1) (fe e2)
        fe (Neg e)     = neg (fe e)
        fe (Num n)     = num n
        fe (Var x)     = var x
        fe (Def d e)   = def (fd d) (fe e)
        fd (Dcl x e)   = dcl x (fe e)
```

```
fe :: E → e
fd :: D → d
```

Adapting evaluation (directly)

Question

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

```
evalE :: E → Env → Int
evalE (Add e₁ e₂) env = evalE e₁ env + evalE e₂ env
evalE (Num e)     env = negate (evalE e env)
evalE (Num n)     env = n
evalE (Var x)     env = env ! x
evalE (Def d e)   env = evalE e _

evalD :: D → _
evalD (Dcl x e)       = _
```

Adapting evaluation (directly)

Question

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

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

Adapting evaluation (as a fold)

```
evalAlgebra :: EDAlgebra (Env → Int) (Env → Env)
evalAlgebra =
  (λe₁ e₂ → λenv → e₁ env + e₂ env,
   λe     → λenv → negate (e env),
   λn     → λenv → n,
   λx     → λenv → env ! x,
   λd e   → λenv → e (d env),
   λx e   → λenv → insert x (e env) env)
```

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
```

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 → E → 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:

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

Adapting evaluation

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

```
evalAlgebra :: EDAlgebra (Env → Int) (Env → Env)
evalAlgebra =
  (λe1 e2 → λenv → e1 env + e2 env,
   λe     → λenv → negate (e env),
   λn     → λenv → n,
   λx     → λenv → env ! x,
   λds e  → λenv → e (process ds env),
   λx e   → λenv → insert x (e env) env)
```

```
process :: [Env → Env] → Env → Env
process ds env = foldl (flip ($)) env ds
```

7.6 Use before definition

Recursion revisited

We said before that we interpret

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

as a redefinition because recursive functions were not useful.

Recursion revisited

We said before that we interpret

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

as a redefinition because recursive functions were not useful.

Let us now reconsider this decision and allow

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

which will evaluate to 8.

RepMax

Simpler example: Let's replace all the elements of a list with the largest number

$$\begin{array}{l}
\text{maxAlg} :: \text{ListAlgebra Int Int} \\
\text{maxAlg} = \text{LAlg}\ \{\text{nil} = \text{minBound}, \text{cons x m} = \text{x `maximum` m}\} \\
\text{repAlg} :: \text{Int} \to \text{ListAlgebra Int [Int]} \\
\text{repAlg m} = \text{LAlg}\ \{\text{nil} = [\,], \text{cons _ xs} = \text{m} : \text{xs}\} \\
\text{repMax xs} = \text{foldr repAlg (foldr maxAlg xs) xs}
\end{array}$$

RepMax

Simpler example: Let's replace all the elements of a list with the largest number

```
maxAlg :: ListAlgebra Int Int
maxAlg = LAlg {nil = minBound, cons x m = x `maximum` m}
repAlg :: Int → ListAlgebra Int [Int]
repAlg m = LAlg {nil = [], cons _ xs = m : xs}
repMax xs = foldr repAlg (foldr maxAlg xs) xs
```

```
repMaxAlg :: ListAlgebra Int (Int → ([Int], Int))
repMaxAlg = LAlg {nil      = λmax → ([], minBound)
                 , cons x f = λmax →
                     let (ys, maxSoFar) = f max
                     in  (max : ys, x `maximum` maxSoFar)}
```

RepMax

```
repMaxAlg :: ListAlgebra Int (Int → ([Int], Int))
repMaxAlg = LAlg { nil        = λmax → ([], minBound)
                 , cons x f = λmax →
                    let (ys, maxSoFar) = f max
                    in  (max : ys, x `maximum` maxSoFar) }
repMax :: [Int] → [Int]
repMax xs = maxs
  where (maxs, max) = foldr repMaxAlg xs max
```

RepMax

```
repMaxAlg :: ListAlgebra Int (Int → ([Int], Int))
repMaxAlg = LAlg {nil        = λmax → ([], minBound)
                 , cons x f = λmax →
                     let (ys, maxSoFar) = f max
                     in  (max : ys, x `maximum` maxSoFar)}

repMax :: [Int] → [Int]
repMax xs = maxs
   where (maxs, max) = foldr repMaxAlg xs max
```

What does 'foldr repMaxAlg xs undefined' return?

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).

Evaluating recursive declarations

The result type for declarations now becomes

> $\text{Env} \rightarrow \text{Env} \rightarrow \text{Env}$

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

$\text{Env} \rightarrow \text{Env} \rightarrow \text{Env}$

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

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

Adapting the fold

```
evalAlgebra :: EDAlgebra (Env → Int) (Env → Env → Env)
evalAlgebra =
  (λe₁ e₂ → λenv      → e₁ env + e₂ env,
   λe     → λenv      → negate (e env),
   λn     → λenv      → n,
   λx     → λenv      → env ! x,
   λds e  → λenv      → let fenv = process ds env fenv
                        in e fenv,
   λx e   → λenv fenv → insert x (e fenv) env)
```

```
process :: [Env → Env → Env] → Env → Env → Env
process ds env fenv = foldl (λcenv d → d cenv fenv) env ds
```

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: regular languages.