



Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Talen en Compilers

2019 - 2020, period 2

Jurriaan Hage

Department of Information and Computing Sciences
Utrecht University

2019-12-05

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:

$\text{eval} :: E \rightarrow \text{Int}$

$\text{eval} (\text{Add } e_1 \ e_2) = \text{eval } e_1 + \text{eval } e_2$

$\text{eval} (\text{Neg } e) = \text{negate } (\text{eval } e)$

$\text{eval} (\text{Num } n) = n$



Evaluation

Directly:

$\text{eval} :: E \rightarrow \text{Int}$

$\text{eval} (\text{Add } e_1 \ e_2) = \text{eval } e_1 + \text{eval } e_2$

$\text{eval} (\text{Neg } e) = \text{negate } (\text{eval } e)$

$\text{eval} (\text{Num } n) = n$

Using foldE:

$\text{evalAlgebra} :: E\text{Algebra } \text{Int}$

$\text{evalAlgebra} = ((+), \text{negate}, \text{id})$

$\text{eval} :: E \rightarrow \text{Int}$

$\text{eval} = \text{foldE } \text{evalAlgebra}$



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

| $\text{Env} \rightarrow \text{Int}$

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:

$\text{eval} :: E \rightarrow \text{Env} \rightarrow \text{Int}$

$\text{eval} (\text{Add } e_1 \ e_2) \ \text{env} = \text{eval } e_1 \ \text{env} + \text{eval } e_2 \ \text{env}$

$\text{eval} (\text{Neg } e) \ \text{env} = \text{negate } (\text{eval } e \ \text{env})$

$\text{eval} (\text{Num } n) \ \text{env} = n$

$\text{eval} (\text{Var } x) \ \text{env} = \text{env} ! x$



Evaluation with variables

Directly:

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

Algebra:

$$\begin{aligned} \text{evalAlgebra} &:: E\text{Algebra} (\text{Env} \rightarrow \text{Int}) \\ \text{evalAlgebra} &= \\ &(\lambda r_1 \ r_2 \rightarrow \lambda \text{env} \rightarrow r_1 \ \text{env} + r_2 \ \text{env}, \\ &\lambda r \quad \rightarrow \lambda \text{env} \rightarrow \text{negate} (r \ \text{env}), \\ &\lambda n \quad \rightarrow \lambda \text{env} \rightarrow n, \\ &\lambda x \quad \rightarrow \lambda \text{env} \rightarrow \text{env} ! x) \end{aligned}$$


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

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 changes 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 changes 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, def) = 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:

$\text{eval} :: E \rightarrow \text{Env} \rightarrow \text{Int}$

... -- as before

$\text{eval} (\text{Def } x \ e_1 \ e_2) \ \text{env} = \text{eval } e_2 \ (\text{insert } x \ (\text{eval } e_1 \ \text{env}) \ \text{env})$

- ▶ 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.



Evaluating expressions with definitions

Directly:

$\text{eval} :: E \rightarrow \text{Env} \rightarrow \text{Int}$

... -- as before

$\text{eval} (\text{Def } x \ e_1 \ e_2) \ \text{env} = \text{eval } e_2 \ (\text{insert } x \ (\text{eval } e_1 \ \text{env}) \ \text{env})$

- ▶ 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:

$\text{evalAlgebra} :: E\text{Algebra} \ (\text{Env} \rightarrow \text{Int})$

$\text{evalAlgebra} = (\dots, \text{ -- as before}$

$\lambda x \ r_1 \ r_2 \rightarrow \lambda \text{env} \rightarrow r_2 \ (\text{insert } x \ (r_1 \ \text{env}) \ \text{env}))$



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?



Algebra for families of datatypes

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

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

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

```
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:

```
foldE :: EDAlgebra e d → Expr → 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)
```



Fold for families of datatypes

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

```
foldE :: EDAlgebra e d → Expr → 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)
```



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?

$\text{evalE} :: E \rightarrow \text{Env} \rightarrow \text{Int}$

$\text{evalE} (\text{Add } e_1 \ e_2) \ \text{env} = \text{evalE } e_1 \ \text{env} + \text{evalE } e_2 \ \text{env}$

$\text{evalE} (\text{Num } e) \ \text{env} = \text{negate} (\text{evalE } e \ \text{env})$

$\text{evalE} (\text{Num } n) \ \text{env} = n$

$\text{evalE} (\text{Var } x) \ \text{env} = \text{env} ! x$

$\text{evalE} (\text{Def } d \ e) \ \text{env} = \text{evalE } e \ (\text{evalD } d \ \text{env})$

$\text{evalD} :: D \rightarrow \text{Env} \rightarrow \text{Env}$

$\text{evalD} (\text{Dcl } x \ e) \ \text{env} = \text{insert } x \ (\text{evalE } e \ \text{env}) \ \text{env}$



Adapting evaluation (as a fold)

$\text{evalAlgebra} :: \text{EDAlgebra} (\text{Env} \rightarrow \text{Int}) (\text{Env} \rightarrow \text{Env})$

$\text{evalAlgebra} =$

$(\lambda e_1 e_2 \rightarrow \lambda \text{env} \rightarrow e_1 \text{ env} + e_2 \text{ env},$

$\lambda e \rightarrow \lambda \text{env} \rightarrow \text{negate } (e \text{ env}),$

$\lambda n \rightarrow \lambda \text{env} \rightarrow n,$

$\lambda x \rightarrow \lambda \text{env} \rightarrow \text{env} ! x,$

$\lambda d e \rightarrow \lambda \text{env} \rightarrow e (d \text{ env}),$

$\lambda x e \rightarrow \lambda \text{env} \rightarrow \text{insert } x (e \text{ env}) \text{ env})$



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 → 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 $\text{Env} \rightarrow \text{Env}$ functions (one for each declaration) in the case for Def:

```
evalAlgebra :: EDAAlgebra (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)
```



Adapting evaluation

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

```
evalAlgebra :: EDAAlgebra (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.



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).



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

$\text{evalAlgebra} :: \text{EDAlgebra} (\text{Env} \rightarrow \text{Int}) (\text{Env} \rightarrow \text{Env} \rightarrow \text{Env})$

$\text{evalAlgebra} =$

$(\lambda e_1 e_2 \rightarrow \lambda \text{env} \rightarrow e_1 \text{ env} + e_2 \text{ env},$

$\lambda e \rightarrow \lambda \text{env} \rightarrow \text{negate } (e \text{ env}),$

$\lambda n \rightarrow \lambda \text{env} \rightarrow n,$

$\lambda x \rightarrow \lambda \text{env} \rightarrow \text{env} ! x,$

$\lambda ds e \rightarrow \lambda \text{env} \rightarrow \text{let } f\text{env} = \text{process } ds \text{ env } f\text{env}$
 $\text{in } e \text{ fenv},$

$\lambda x e \rightarrow \lambda \text{env } f\text{env} \rightarrow \text{insert } x \text{ (} e \text{ fenv) env}$)



Adapting the fold

$\text{evalAlgebra} :: \text{EDAlgebra} (\text{Env} \rightarrow \text{Int}) (\text{Env} \rightarrow \text{Env} \rightarrow \text{Env})$

$\text{evalAlgebra} =$

$(\lambda e_1 e_2 \rightarrow \lambda \text{env} \rightarrow e_1 \text{ env} + e_2 \text{ env},$

$\lambda e \rightarrow \lambda \text{env} \rightarrow \text{negate } (e \text{ env}),$

$\lambda n \rightarrow \lambda \text{env} \rightarrow n,$

$\lambda x \rightarrow \lambda \text{env} \rightarrow \text{env} ! x,$

$\lambda ds e \rightarrow \lambda \text{env} \rightarrow \text{let } f\text{env} = \text{process } ds \text{ env } f\text{env}$
 $\text{in } e \text{ fenv},$

$\lambda x e \rightarrow \lambda \text{env } f\text{env} \rightarrow \text{insert } x \text{ (} e \text{ fenv) env}$)

$\text{process} :: [\text{Env} \rightarrow \text{Env} \rightarrow \text{Env}] \rightarrow \text{Env} \rightarrow \text{Env} \rightarrow \text{Env}$

$\text{process } ds \text{ env } f\text{env} = \text{foldl } (\lambda c\text{env } d \rightarrow d \text{ cenv } f\text{env}) \text{ env } ds$



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.

