The Architecture of the Utrecht Haskell Compiler

Atze Dijkstra, Jeroen Fokker, Doaitse Swierstra Utrecht University

HS09, September 3, 2009

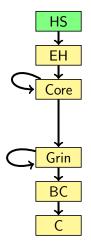
Views on UHC

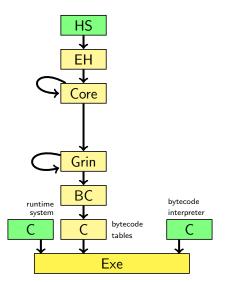
- From input to output
- Specification of "From input to output"

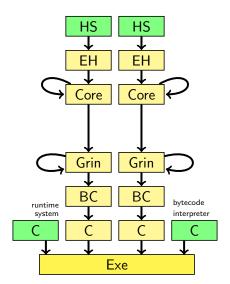
Views on UHC

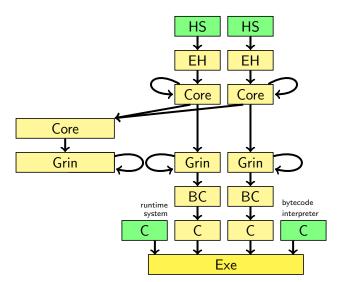
- From input to output
 Pipeline of transformations on internal representations
- Specification of "From input to output"
 Aspectwise organisation of code
 Tools

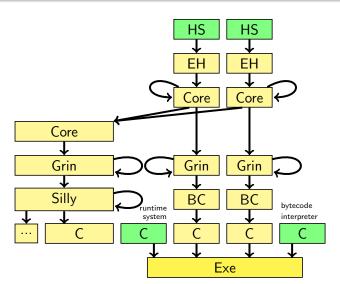
HS

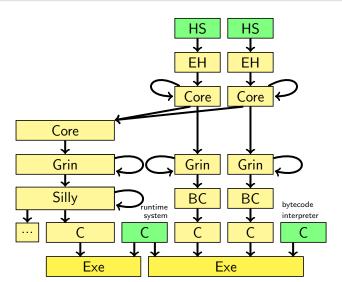


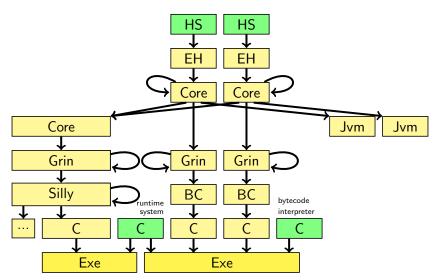


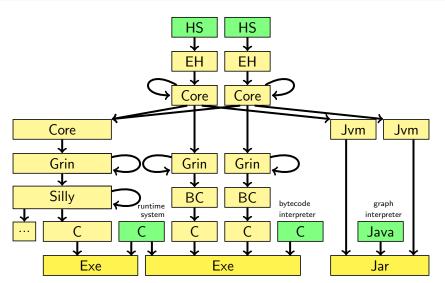












Tools for UHC implementation

- Functional programming
- Tree-oriented programming
- Rule-oriented programming
- Aspect-oriented programming

Tools for UHC implementation

- Functional programming
 GHC: Haskell compiler
- Tree-oriented programming
 UUAG: Attribute Grammar preprocessor
- Rule-oriented programming
 Ruler: Type rule preprocessor
- Aspect-oriented programming
 Shuffle: Source fragmentation preprocessor

Tools for UHC implementation

- Functional programming GHC: Haskell compiler
- Tree-oriented programming
 UUAG: Attribute Grammar preprocessor
- Rule-oriented programming Ruler: Type rule preprocessor
- Aspect-oriented programming
 Shuffle: Source fragmentation preprocessor

data Expr

=

Con Int Add Expr Expr Mul Expr Expr



data Expr

 $calc :: Expr \rightarrow Int$

=

Con Int Add Expr Expr Mul Expr Expr

data Expr	fold	$calc :: Expr \rightarrow Int$
= Con Int Add Expr Expr Mul Expr Expr	$ \begin{array}{c} :: \\ (Int \rightarrow b) \\ \rightarrow (b \rightarrow b \rightarrow b) \\ \rightarrow (b \rightarrow b \rightarrow b) \\ \rightarrow Expr \rightarrow b \end{array} $	

$$\begin{array}{lll} \textbf{data } \textit{Expr} & \textbf{type } \textit{Sem } \textit{b} & \textit{calc} :: \textit{Expr} \rightarrow \textit{Int} \\ & = & & \textit{calc} = \textit{fold} \\ & (& (\textit{Int} \rightarrow \textit{b}) & (\lambda \textit{n} \rightarrow \textit{n} &) \\ & | & \textit{Add } \textit{Expr } \textit{Expr} & , & (\textit{b} \rightarrow \textit{b} \rightarrow \textit{b}) & (\lambda \textit{x} \textit{y} \rightarrow \textit{x} + \textit{y}) \\ & | & \textit{Mul } \textit{Expr } \textit{Expr} & , & (\textit{b} \rightarrow \textit{b} \rightarrow \textit{b}) & (\lambda \textit{x} \textit{y} \rightarrow \textit{x} * \textit{y}) \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \textit{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \textit{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b} \rightarrow \text{otherwise} \\ & | & \text{fold } :: \textit{Sem } \textit{b}$$

```
data Expr
                                  type Sem b
                                                                   calcsem :: Sem Int.
                                                                   calcsem =
                                   ( \ (\textit{Int} \quad \to b) \quad \Big| \ (\lambda n \quad \to n
     Con Int
     Add Expr Expr \Big| \ , \ (b \rightarrow b \rightarrow b) \ \Big| \ , \lambda x \ y \rightarrow x + y
     Mul Expr Expr \left[ \begin{array}{c} , \ (b \rightarrow b \rightarrow b) \end{array} \right], \lambda x \ y \rightarrow x * y
                                    , (Name \rightarrow b) \mid)
     Var Name
                                  fold :: Sem b \rightarrow | calc :: Expr \rightarrow Int
                                             Expr \rightarrow b | calc = fold \ calcsem
```

```
data Expr
                             type Sem b
                                                          calcsem :: Sem Int.
                                                          calcsem =
                              ( (Int \rightarrow b) \mid (\lambda n \rightarrow n)
    Con Int
    Add Expr Expr | \ , \ (b \rightarrow b \rightarrow b) \ | \ , \lambda x \ y \rightarrow x + y
    Mul Expr Expr (b \rightarrow b \rightarrow b) \lambda x y \rightarrow x * y
                               , (Name \rightarrow b) \mid , \lambda s \rightarrow lookup s e
    Var Name
                             fold :: Sem b \rightarrow | calc :: Expr \rightarrow Int
                                       Expr \rightarrow b | calc = fold \ calcsem
```

type Sem b calcsem :: Sem Int. data Expr calcsem = ((Int \rightarrow b) $|(\lambda n \rightarrow n)|$ Con Int Add Expr Expr $| \ , \ (b \rightarrow b \rightarrow b) \ | \ , \lambda x \ y \rightarrow x + y$ Mul Expr Expr $(b \rightarrow b \rightarrow b)$ $\lambda x y \rightarrow x * y$, $(Name \rightarrow b)$ $\lambda s \rightarrow \lambda e \rightarrow lookup s e$ Var Name $fold :: Sem b \rightarrow | calc :: Expr \rightarrow Int$ $Expr \rightarrow b$ | $calc = fold \ calcsem$

```
type Sem b
data Expr
                                                           calcsem :: Sem (Env \rightarrow Int)
                                                           calcsem =
                               ((Int \rightarrow b) | (\lambda n \rightarrow n)
    Con Int
    Add Expr Expr |  , (b \rightarrow b \rightarrow b)  | , \lambda x \ y \rightarrow x + y
    Mul Expr Expr | , (b \rightarrow b \rightarrow b) | , \lambda x y \rightarrow x * y
                                , (Name \rightarrow b) \lambda s \rightarrow \lambda e \rightarrow lookup s e
    Var Name
                              fold :: Sem b \rightarrow | calc :: Expr \rightarrow Int
                                       Expr \rightarrow b | calc = fold \ calcsem
```

```
type Sem b
                                                                 calcsem :: Sem (Env \rightarrow Int)
data Expr
                                                                 calcsem =
     Con Int
                                   ((Int \rightarrow b) | (\lambda n \rightarrow \lambda e \rightarrow n)
    Add Expr Expr | \ , \ (b \rightarrow b \rightarrow b) \ | \ , \lambda x \ y \rightarrow \lambda e \rightarrow x \ e + y \ e
     Mul Expr Expr | , (b \rightarrow b \rightarrow b) | , \lambda x y \rightarrow \lambda e \rightarrow x e * y e
                                   , (Name \rightarrow b) \mid , \lambda s \rightarrow \lambda e \rightarrow lookup s e
     Var Name
                                 fold :: Sem b \rightarrow | calc :: Expr \rightarrow Int
                                           Expr \rightarrow b | calc = fold \ calcsem
```


 $Expr \rightarrow b$ | $calc = fold \ calcsem \ testenv$

type Sem b data Expr calcsem :: Sem (Env \rightarrow Int) ^C Inherited $((Int \rightarrow b) \mid (] attribute) \rightarrow r$ Synthesized Con Int Add Expr Expr $| \ , \ (b \rightarrow b \rightarrow b) \ | \ , \lambda x \ y \rightarrow \lambda e \rightarrow x$ attribute Mul Expr Expr $| , (b \rightarrow b \rightarrow b) | , \lambda x y \rightarrow \lambda e \rightarrow x e * y e$, $(Name \rightarrow b)$ $\lambda s \rightarrow \lambda e \rightarrow lookup s e$ Var Name

fold :: Sem
$$b \rightarrow Expr \rightarrow b$$

 $\mid calc :: Expr \rightarrow Int$ $Expr \rightarrow b$ | $calc = fold \ calcsem \ testenv$

data Exprtype Sem b $calcsem :: Sem (Env <math>\rightarrow Int)$)=Con Int $(Int \rightarrow b)$ Synthesized| Add Expr Expr $(b \rightarrow b \rightarrow b)$ $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | Aul Expr Expr $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$ | $Ax y \rightarrow \lambda e \rightarrow x e * y e$

 $Expr \rightarrow b$ | $calc = fold \ calcsem \ testenv$

data Expr

=

Con con: Int

Add lef: Expr rit: Expr Mul lef: Expr rit: Expr

Var name : Name

Named fields

```
calcsem :: Sem (Env \rightarrow Int)

Continuous Inherited (attribute) \rightarrow r Synthesized attribute

\lambda x \ y \rightarrow \lambda e \rightarrow x \ e * y \ e

\lambda x \ y \rightarrow \lambda e \rightarrow x \ e * y \ e

\lambda x \ y \rightarrow \lambda e \rightarrow lookup \ s \ e
```

```
data Expr
                                                       calcsem :: Sem (Env \rightarrow Int)
                                                       C. Inherited
                                                       (r attribute r \rightarrow r Synthesized
    Con con: Int
                                                       \lambda x y \rightarrow \lambda e \rightarrow x attribute
    Add lef: Expr rit: Expr
                                                       \lambda x y \rightarrow \lambda e \rightarrow x e * y e
    Mul lef: Expr rit: Expr
                                                       \lambda s \rightarrow \lambda e \rightarrow lookup s e
    Var name: Name
                                        Named
                                       attributes
  Named
                            attr Expr inh env : Env
   fields
                                          syn val: Int
```

```
data Expr
                                                   calcsem :: Sem (Env \rightarrow Int)
                                                   C. Inherited
                                                   (r attribute r \rightarrow r Synthesized
    Con con: Int
                                                   \lambda x y \rightarrow \lambda e \rightarrow x attribute
    Add lef: Expr rit: Expr
                                                   \lambda x y \rightarrow \lambda e \rightarrow x e * y e
    Mul lef: Expr rit: Expr
                                                   \lambda s \rightarrow \lambda e \rightarrow lookup s e
    Var name: Name
                                      Named
                                    attributes
  Named
                          attr Expr inh env : Env
   fields
                                       syn val: Int
                          sem Expr | Mul lhs.val = @lef.val * @rit.val
                                               lef env = 0lhs env
                                               rit.env = @lhs.env
```

Attribute Grammar processor

UUAG Attribute Grammar preprocessor lets you

- have named fields and attributes
- define semantics by defining attributes

and automatically

- generates the fold-function
- generates the semantic functions to instantiate it
- inserts trivial rules

AG applied in UHC

Desired Core transformation:

Inline name aliases

AG applied in UHC

Desired Core transformation:

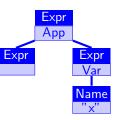
Inline name aliases

- Gather introduced bindings bottom up
- Distribute gathered bindings top down
- Compute transformed tree bottom up, variable occurrences possibly replaced

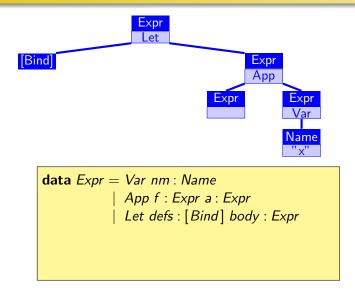
data Expr = Var nm : Name

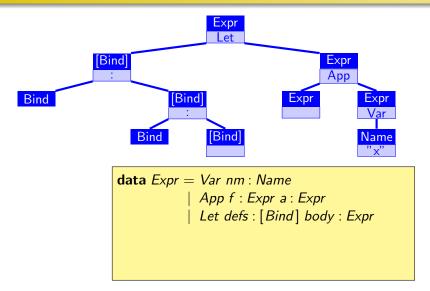


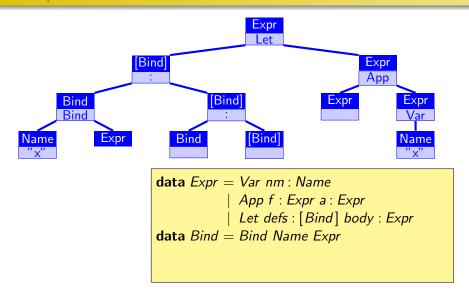
data Expr = Var nm : Name

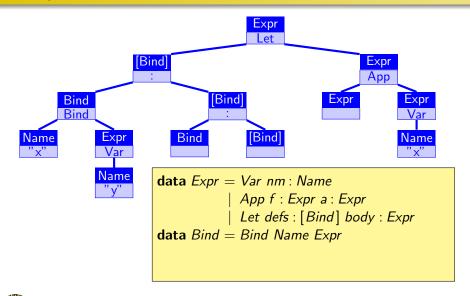


data Expr = Var nm : Name | App f : Expr a : Expr

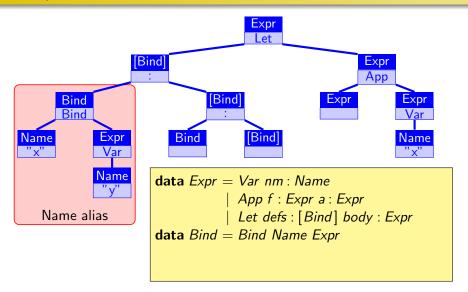


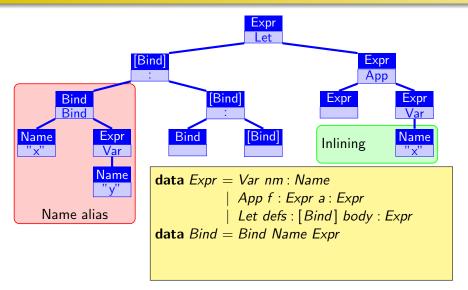


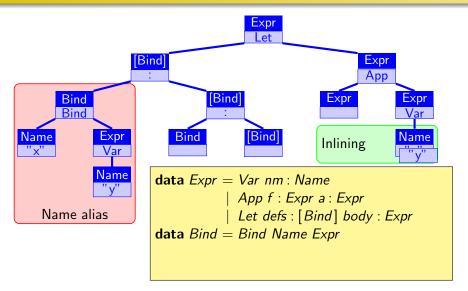


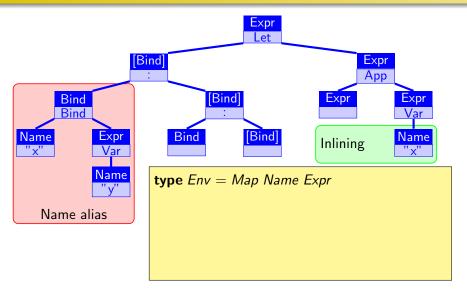


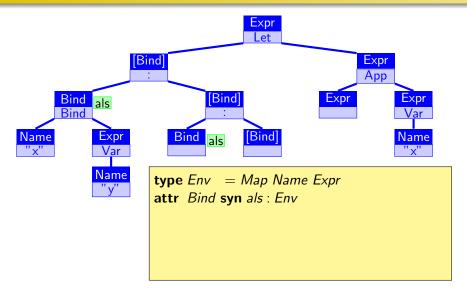
Example Core transformation: Name alias

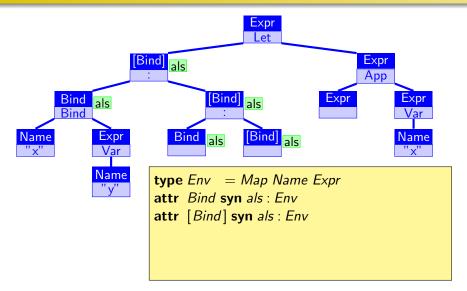


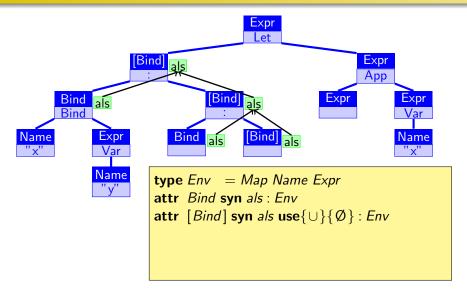


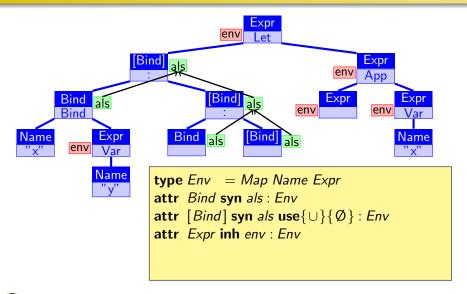


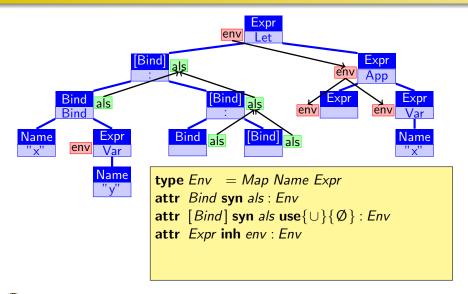


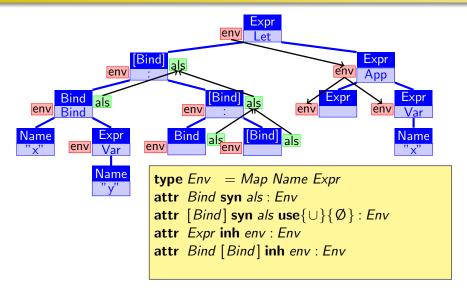


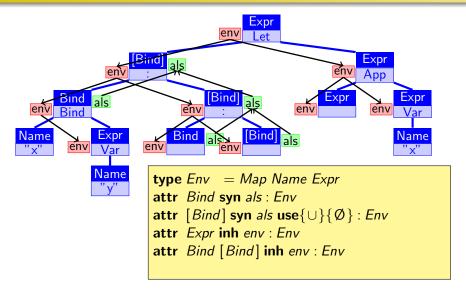


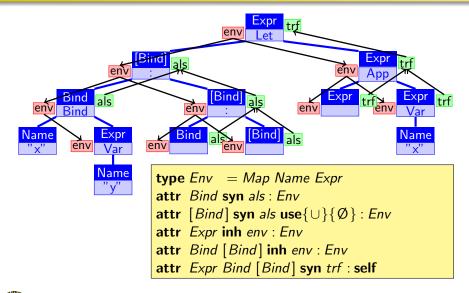




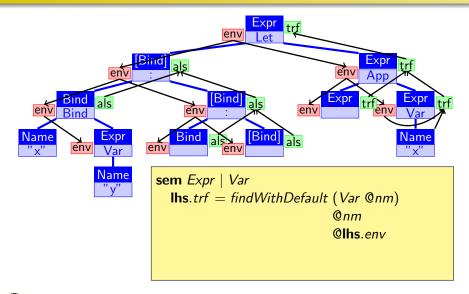


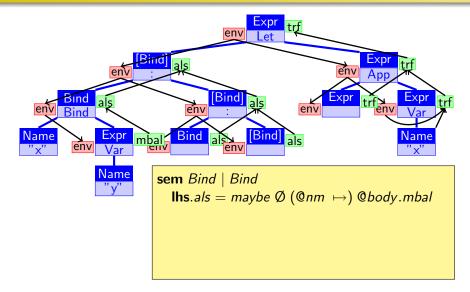


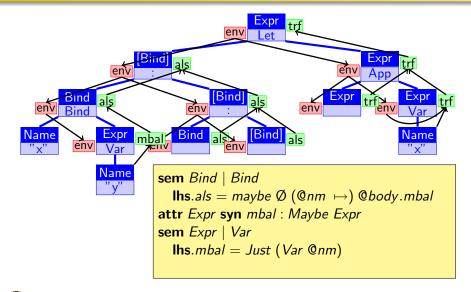


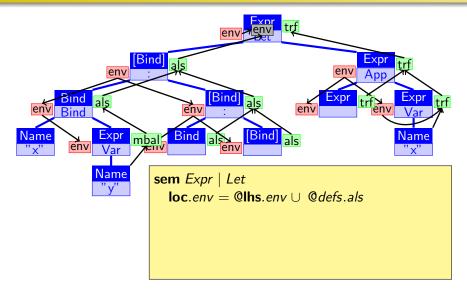


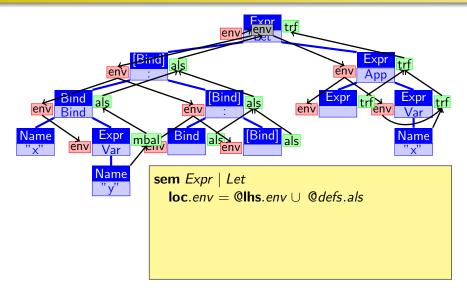


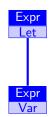




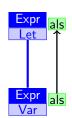




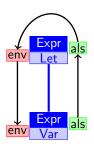




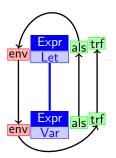
- First pass
- Second pass



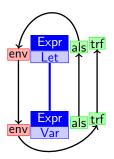
- First pass
 Bottom-up gather *aliases*
- Second pass



- First pass
 Bottom-up gather aliases
- Second pass
 Top-down distribute environment



- First pass
 Bottom-up gather aliases
- Second pass
 Top-down distribute environment
 Bottom-up generate transformed tree



Name alias inlining is a two-pass traversal:

- First pass
 Bottom-up gather aliases
- Second pass
 Top-down distribute environment
 Bottom-up generate transformed tree

Either rely on lazyness
Or let UUAG schedule the passes
(and do cycle check)

AG programming in UHC

Attribute Grammar descriptions for

Analyses

HS: Name dependency

EH: Typing

Grin: Heap Points To, Call Graph

Transformations

Core: Lambda Lifting

Grin: Drop Unreachable Bindings

Generation

Next language, Pretty Printing

AG programming in UHC

Attribute Grammar descriptions for

Analyses

HS: Name dependency

EH: Typing

Grin: Heap Points To, Call Graph

Transformations

Core: Lambda Lifting

Grin: Drop Unreachable Bindings

Generation

Next language, Pretty Printing

datas, attrs, and sems grouped as aspects

and actually...



Aspectwise organisation

All code grouped as aspects

Along two dimensions:

- Variant: a step in the stepwise growth in language features
- Aspect: a set of attributes and its resulting computation

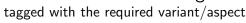
Aspectwise organisation

All code grouped as aspects

Along two dimensions:

- Variant: a step in the stepwise growth in language features
- Aspect: a set of attributes and its resulting computation

Shuffle extracts code fragments tagged with the required variant/aspect



Variants and aspects

E.g. aspect *typing* for the first variants of UHC:

• variant 1: simply typed λ -calculus i :: Int i=5

Variants and aspects

E.g. aspect *typing* for the first variants of UHC:

- variant 1: simply typed λ -calculus i::Int i=5
- variant 3: polymorphic type inference
 id x = x
 infers id :: ∀a.a → a

E.g. aspect *typing* for the first variants of UHC:

- variant 1: simply typed λ -calculus i::Int i=5
- variant 3: polymorphic type inference
 id x = x
 infers id :: ∀a.a → a
- variant 4: higher ranked types
 f:: (∀a.a → a) → (Char, Int)
 f i = (i 'x', i 5)
 v = f id
 allows polymorphic use of argument i inside f

Variants

In UHC

- \bigcirc λ -calculus, type checking
- type inference
- polymorphism
- 4
- data types

higher ranked types, existentials

Variants

In UHC

plain Haskell	experiments
$oldsymbol{0}$ λ -calculus, type checking	
type inference	
polymorphism	
4	higher ranked types, existentials
data types	
kind inference	kind signatures
records	tuples as records
code generation	full program analysis
classes, type-synonyms	extensible records

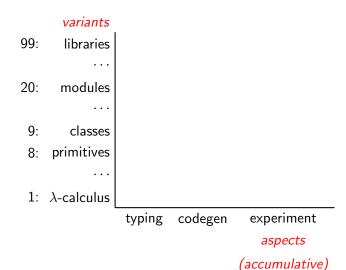
Variants

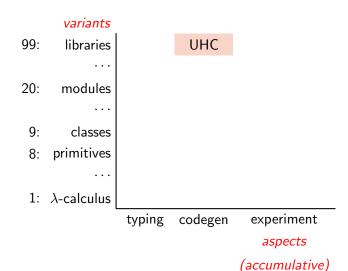
In UHC

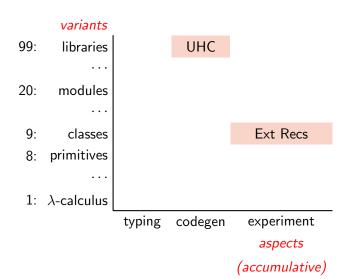
plain Haskell	experiments
() λ -calculus, type checking	
type inference	
polymorphism	
3	higher ranked types, existentials
data types	
kind inference	kind signatures
records	tuples as records
code generation	full program analysis
classes, type-synonyms	extensible records
modules	
'deriving'	exceptions

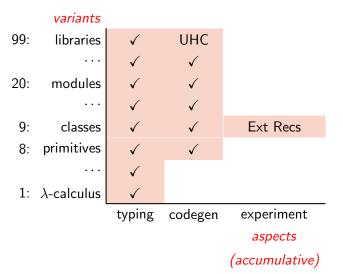


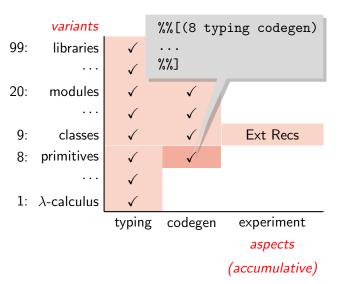
prelude, I/O





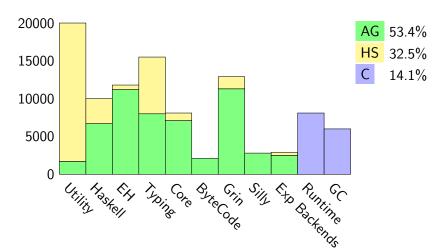






Metrics

We use Attribute Grammars a lot!



Summary

Tree orientation:
Attribute Grammar

Conciseness Attribute specification Boilerplate generation - trivial copies

New idiom

- fold

Summary

Tree orientation:
Attribute Grammar

Aspect orientation:

Shuffle

Conciseness
Attribute specification

Boilerplate generation

- trivial copies
- fold

Variant:

feature isolation

Aspect:

experimentation

New idiom

Fixed order

Summary

Tree orientation:
Attribute Grammar

Aspect orientation: Shuffle General

Conciseness

Attribute specification Boilerplate generation

- trivial copies
- fold

Variant:

feature isolation

Aspect:

 ${\sf experimentation}$

Simpler design and coding

New idiom

Fixed order

Complex build system