Utrecht Haskell Compiler

Atze Dijkstra, Jeroen Fokker, Doaitse Swierstra
Utrecht University

AFP 2010, Aug 25

Utrecht Haskell Compiler (UHC)

Is a Haskell compiler (obviously)

Is a compiler & language experimentation platform

Is an engineering challenge to deal with complexity

Utrecht Haskell Compiler (UHC)

Is a Haskell compiler (obviously)

- Most of Haskell98, Haskell2010
- Extensions (higher ranked types, polymorphic kinds)
- Multiple backends
- Slowly matures towards usable tool

Is a compiler & language experimentation platform

- Whole program analysis
- Type system

Is an engineering challenge to deal with complexity

- Tree-oriented programming: Attribute Grammar system (AG)
- DSLs for subproblems: aspectwise organisation, type system specification
- Divide & conquer: into aspects, into isolated problems, into transformations, ...

Todays story

Understand how functional program transforms to runnable program

Compiler itself is case study of functional programming

Todays story

Understand how functional program transforms to runnable program

- Pipeline of transformations
- Relation between programming luxury and implementation price

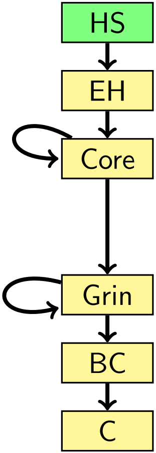
Compiler itself is case study of functional programming

- Folds over abstract syntax tree representation
- Attribute grammar system

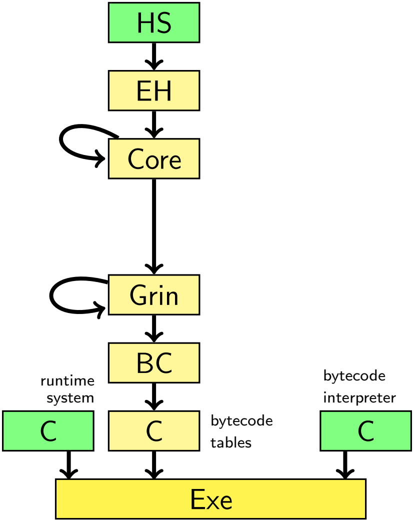
UHC pipeline

HS

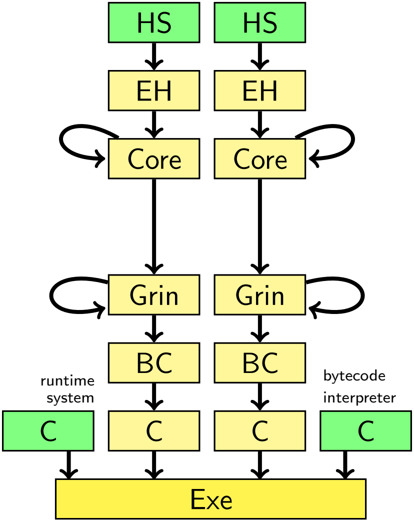
UHC pipeline



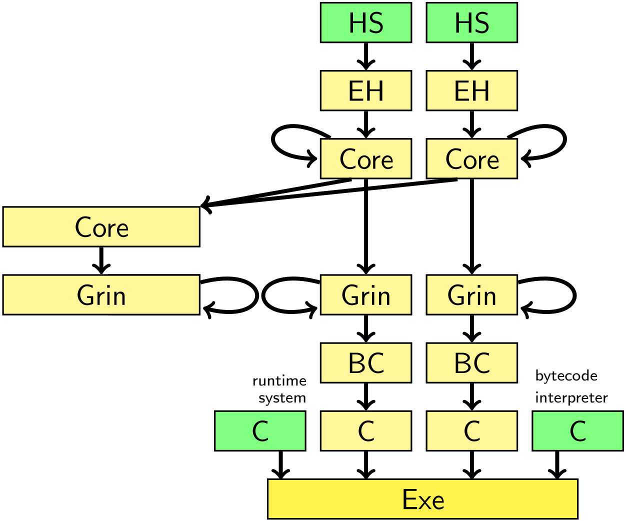
UHC pipeline



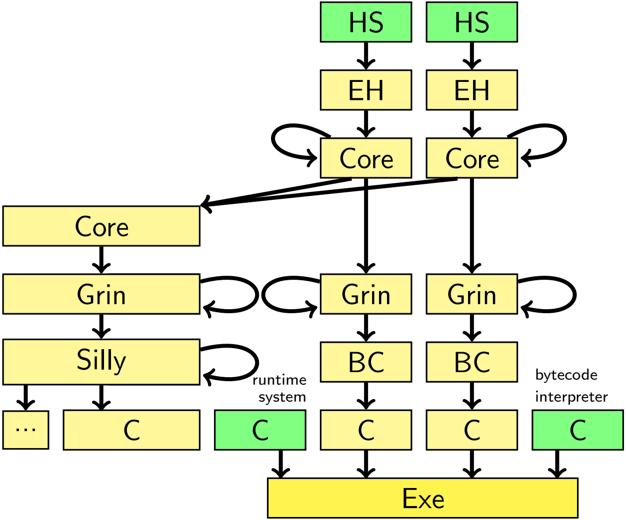
UHC pipeline



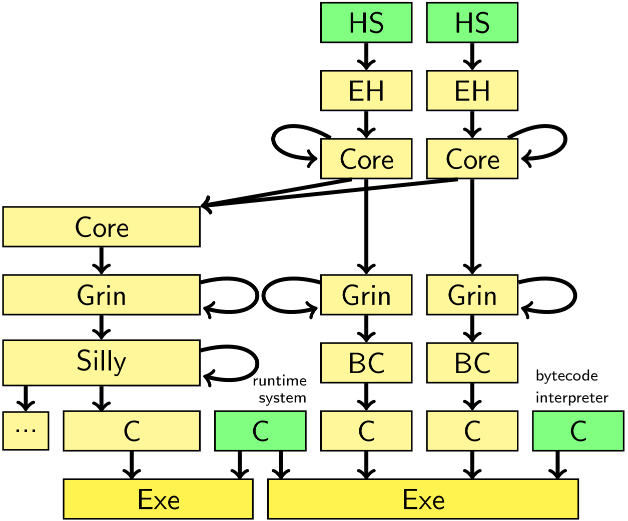
UHC pipeline



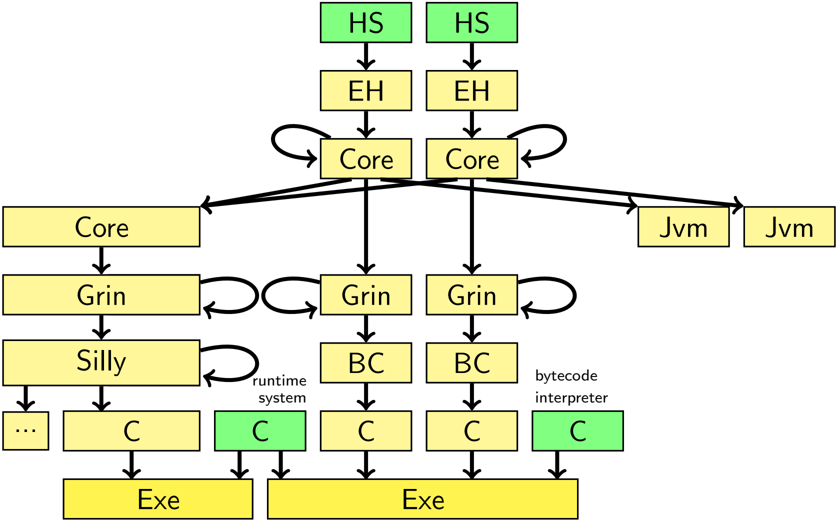
UHC pipeline



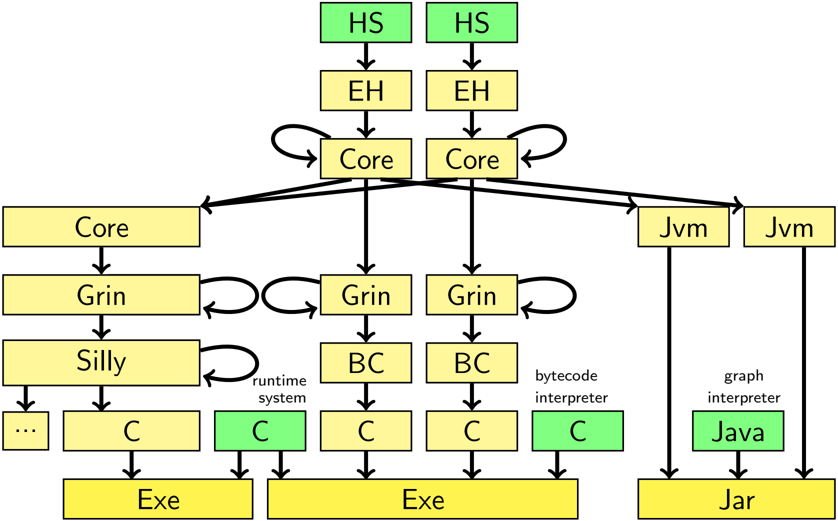
UHC pipeline



UHC pipeline



UHC pipeline



Pipeline running example

module *Main* **where**

$len :: [a] \rightarrow Int$
$len\ [\,] = 0$
$len\ (x : xs) = 1 + len\ xs$

$main = putStr\ (show\ (len\ (replicate\ 4\ \texttt{'x'})))$

Pipeline: desugaring to “Essential Haskell”

```
-- EH
let vb1.len :: [a] → Int
    vb1.len
      = λx1 → case x1 of
            EHC.Prelude.[]
              → EHC.Prelude.fromInteger 0
            (EHC.Prelude.: x xs)
              → ...
in
let vb1.main = EHC.Prelude.putStr ...
in
let main :: EHC.Prelude.IO ...
    main = EHC.Prelude.ehcRunMain vb1.main
in
main
```

```
len :: [a] → Int
len [] = 0
len (x : xs) = 1 + len xs
```

Pipeline: desugaring

Name resolution

Binding groups

Syntactic sugar

Pipeline: desugaring

Name resolution

- To which definition from which module refers an identifier?
- Replace by explicit qualified reference

Binding groups

- Subsequent (type) analysis requires “define before use”
- Replace by ordered strongly connected components (based on dependency graph)

Syntactic sugar

- Alternate syntax for similar semantics: duplicate work later on
- Replace by simpler constructs:

$$
\begin{array}{lcl}
\textbf{do}\ \{ s_1; ..\ s_2 \} & \leadsto & s_1 \ggg= ..\ s_2 \\
e\ \textbf{where}\ d & \leadsto & \textbf{let}\ d\ \textbf{in}\ e \\
\textbf{if}\ c\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \leadsto & \textbf{case}\ c\ \textbf{of}\ \{ \mathit{True} \rightarrow e_1; \mathit{False} \rightarrow e_2 \} \\
f\ p_1 = e_1; f\ p_2 = e_2 & \leadsto & \lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ \{ p_1 \rightarrow e_1; p_2 \rightarrow e_2 \}
\end{array}
$$

Pipeline: type directed translation to untyped Core

```
-- Core
module $vb1 =
let rec
  {$vb1$.len =
    λ$vb1$.x1__1 →
      let !
        {$vb1$.4_42_0$!__3_0 =
          $vb1$.x1__1} in
      case $vb1$.4_42_0$!__3_0 of
        {{ 0, 2, 2 } {..., ...} →
            ...
        ; { 1, 0, 2 } { } →
            let
              {$13_0_14 =
                ($EHC$.Prelude$.packedStringToInteger)
                  (#String "0")} in
            let
              {$13_0_12 =
                ($EHC$.Prelude$.fromInteger)
                  ($EHC$.Prelude$.3_237_0_instance_Num )
                  ($13_0_14)} in
            $13_0_12
        }
in ...
```

```
len :: [a] → Int
len [] = 0
len (x : xs) = 1 + len xs
```

Core: untyped lambda calculus (+ ...)

Core: basics

$$
\begin{array}{lcll}
e & ::= & x & \text{-- variable} \\
 & \mid & \mathit{int} \mid \mathit{char} \mid \mathit{string} \mid \mathit{integer} & \text{-- literal} \\
 & \mid & \mathit{CTag} & \text{-- constructor tag} \\
 & \mid & \textbf{let}\ [b]\ \textbf{in}\ e & \text{-- binding} \\
 & \mid & \textbf{letrec}\ [b]\ \textbf{in}\ e & \text{-- recursive binding} \\
 & \mid & \textbf{let}\ !\ [b]\ \textbf{in}\ e & \text{-- strict binding} \\
 & \mid & \lambda x \rightarrow e & \text{-- abstraction} \\
 & \mid & e_1\ e_2 & \text{-- application} \\
 & \mid & \textbf{case}\ e\ \textbf{of}\ [a] & \text{-- inspection} \\
b & ::= & x = e & \text{-- plain binding} \\
 & \mid & \textbf{ffi}\ \mathit{ccall}\ \texttt{"x"}\ x & \text{-- ffi binding} \\
a & ::= & p \rightarrow e & \text{-- case alternative} \\
p & ::= & \mathit{CTag}\ [x] & \text{-- constructor pattern} \\
\mathit{prog} & ::= & \textbf{module}\ x = e & \text{-- program}
\end{array}
$$

Pipeline: type analysis

Type inference

Class overloading resolution

Type based code generation

Pipeline: type analysis

Type inference

- Hindley-Milner
- Propagation of type annotations

Class overloading resolution

- Determine instance for class predicate

  $(+)\ 1\ (\mathit{len}\ \mathit{xs}) \ \rightsquigarrow\ ((+) :: \mathit{Num}\ \mathit{Int} \Rightarrow \mathit{Int} \rightarrow \mathit{Int})\ (1 :: \mathit{Int})\ (\mathit{len}\ \mathit{xs} :: [\mathit{I}$

Type based code generation

- Dictionary, deriving, and generics

  $$\begin{array}{lll}
  [3] == [4] & \rightsquigarrow & (==)\ (\mathit{dEqList}\ \mathit{dEqInt})\ [3]\ [4] \\
  \textbf{data}\ D \ldots = \ldots \textbf{deriving}\ C & \rightsquigarrow & \textbf{instance}\ C\ D\ \textbf{where} \ldots \\
  \textbf{data}\ D \ldots & \rightsquigarrow & \textbf{instance}\ \mathit{Representable}\ D \\
   & \rightsquigarrow & \textbf{instance}\ \mathit{Datatype}\ D; \ldots
  \end{array}$$

- Expressed in Core directly

Core: the rest

Delayed code generation

Core: the rest

Delayed code generation

- Overloading resolution delayed (i.e. not following AST)
- "yet to be generated code" must be referred to

$$
\begin{aligned}
x == y &\leadsto (==)\ d\,?\ \ x\ \ y \\
&\leadsto (==)\ d\,?\ (x :: Int)\ (y :: Int) \\
&\leadsto (==)\ dEqInt\ (x :: Int)\ (y :: Int)
\end{aligned}
$$

Core: the rest

Delayed code generation

- Overloading resolution delayed (i.e. not following AST)
- "yet to be generated code" must be referred to

$$\begin{aligned} x == y &\leadsto (==)\ d\,?\quad x\quad y \\ &\leadsto (==)\ d\,?\quad (x :: Int)\ (y :: Int) \\ &\leadsto (==)\ dEqInt\ (x :: Int)\ (y :: Int) \end{aligned}$$

- Holes in code: "code variables"

$$\begin{aligned} e ::= &\ \ .\,. \\ \mid &\ \ UID \quad \text{-- hole, identified by globally unique id} \end{aligned}$$

- ... to be substituted later, before code emission
- Common solution idiom for dealing with "yet unknown"

Core: simplifications

Same syntax, simpler form

Unnecessary mutual recursion

Core: simplifications

Same syntax, simpler form

Unnecessary mutual recursion

- Replace "not really" mutually recursive bindings

  $$\textbf{letrec}\ \{v_1 = ..; v_2 = ..\}\ \textbf{in}\ ..$$

  by

  $$\textbf{let}\ v_1 = ..\,\textbf{in}\ \textbf{let}\ v_2 = ..\,\textbf{in}\ ..$$
- Keeps Core generation simpler

Core: simplifications

Trivial application arguments

- Replace complex function arguments

  $f\ (g\ a)\ (h\ b)$

  by simple variables + extra bindings

  $\textbf{let}\ v_1 = g\ a\ \textbf{in}$
  $\textbf{let}\ v_2 = h\ b\ \textbf{in}$
  $\quad f\ v_1\ v_2$

- Closer to actual code
- Administrative normal (A-normal) form

Core: simplifications

Lambda lifting

- Replace implicit globals

$$\begin{aligned} g = \lambda x\ z \rightarrow \textbf{let}\ & f = \lambda y \rightarrow x + y\ \textbf{in} \\ & f\ z \end{aligned}$$

  by explicit arguments

$$\begin{aligned} f' &= \lambda x\ y \rightarrow x + y \\ g &= \lambda x\ z \rightarrow \textbf{let}\ f = f'\ x\ \textbf{in} \\ &\qquad\qquad\quad f\ z \end{aligned}$$

- No need to deal with environments (in analyses, when constructing closures)
- No local lambdas anymore, lifted to outermost level

Pipeline: lambda lifted translation to lazy-less Grin

```
  -- GRIN
module $vb1
{ rec
   { $vb1.len $vb1.x1__1
      = { eval $vb1.x1__1; λ$vb1.4_42_0!__3_0 →
              case $vb1.4_42_0!__3_0 of
              { (#0/C {2, 2})
                  → {...}
              ; (#1/C {0, 2})
                  → { store (#0/C {1, 1}/$EHC.Prelude.PackedString "0"); λ$19_31_0 →
                      store (#0/F/$EHC.Prelude.packedStringToInteger $19_31_0); λ$13_0_14 →
                      store (#0/P/0/$EHC.Prelude.fromInteger$ EHC.Prelude.3_237_0__Num); λ$19_33_0 →
                      store (#0/A/$_ $19_33_0 $13_0_14); λ$13_0_12 →
                      eval $13_0_12 }
              }}}
```

```
len :: [a] → Int
len [] = 0
len (x : xs) = 1 + len xs
```

- Evaluation explicit
- Laziness explicit
- Starting point for both interpreter and whole program analysis

Grin: basics

$$
\begin{array}{rcll}
e & ::= & \mathbf{unit}\ v & \text{-- basic value for direct use} \\
 & \mid & e; \lambda p \rightarrow e & \text{-- sequencing} \\
 & \mid & \mathbf{eval}\ x & \text{-- evaluation} \\
 & \mid & \mathbf{apply}\ x\ [v] & \text{-- application} \\
 & \mid & \mathbf{store}\ v & \text{-- construct heap cell for node} \\
 & \mid & \mathbf{fetch}\ x & \text{-- retrieve heap cell for node} \\
 & \mid & \mathbf{update}\ x\ v & \text{-- overwrite heap cell} \\
 & \mid & \mathbf{case}\ v\ [a] & \text{-- node inspection} \\
v & ::= & x & \text{-- variable} \\
 & \mid & tag\ [v] & \text{-- node construction} \\
 & \mid & int \mid string & \text{-- literal} \\
p & ::= & x & \text{-- variable (bind)} \\
 & \mid & tag\ [x] & \text{-- node (bind fields)} \\
a & ::= & p \rightarrow e & \text{-- case alternative}
\end{array}
$$

Grin: basics

$$
\begin{array}{lll}
tag & ::= \mathbf{C}\ CTag & \text{-- plain constructor} \\
 & \mid\ \mathbf{F}\ x & \text{-- saturated function closure} \\
 & \mid\ \mathbf{P}\ x\ int & \text{-- non-saturated function closure} \\
 & \mid\ \mathbf{A} & \text{-- apply closure} \\
prog & ::= \mathbf{module}\ x\ [c]\ [b] & \text{-- program} \\
b & ::= x\ [x] = e & \text{-- function binding} \\
c & ::= x = v & \text{-- CAF binding}
\end{array}
$$

- Global bindings are mutual recursive

Grin: explicit eval (and apply)

Eval knows to evaluate

```
$eval $x_5
    = { fetch $x_5; λ$x_1013 →
      case $x_1013 of
        {(#0/C {2, 2}/$$, 2 $x_1132 $x_1133)
            → {unit $x_1013}
        ; (#0/C {1, 1}/$Int $x_1130)
            → {unit $x_1013}
        ..
        ; (#0/P/1/$UHC.Base.primSubInt $x_1125)
            → {unit $x_1013}
        ; (#0/P/2/$UHC.Base.primSubInt)
            → {unit $x_1013}
        ..
        ; (#0/F/$Main.len $x_1035)
            → {call $Main.len $x_1035; λ$x_1036 →
               update $x_1036 $x_5}
        ; (#0/F/$UHC.Base.replicate∼spec1)
            → {call $UHC.Base.replicate∼spec1; λ$x_1051 →
               update $x_1051 $x_5}
        ..
        }}
```

Closed world when doing whole program analysis:

- $**eval** knows how to evaluate *all* nodes occurring in the program
- (and **apply** knows to apply)

Grin: eval inlining

Eval inlining

```
$vb1.len $vb1.x1__1
    = { fetch $vb1.x1__1; λ$x_1009 →
        case $x_1009 of
          { (#0/C {2, 2}/$UHC.Base.$ : $x_1011 $x_1012)
                → { unit $x_1009 }
          ; (#1/C {0, 2}/$UHC.Base.[])
                → { unit $x_1009 }
          ; (#0/F/$UHC.Base.replicate∼spec1)
                → { call $UHC.Base.replicate∼spec1; λ$x_1010 →
                    update $x_1010 $vb1.x1__1 }
          } ; λ ...
          }}
```

Using “Heap Points To” (HPT) analysis

- Inline $**eval** alternatives only for node formats known to be pointed to

Grin: the rest

Foreign functions

- Must be explicit in (un)boxing
- Requires annotation of variables

$$
\begin{array}{lll}
e & ::= & .. \\
  & \mid & \textbf{ffi}\ \mathit{ccall}\ \texttt{"x"}\ x \qquad \text{-- foreign function call} \\
p & ::= & .. \\
  & \mid & \textbf{basic}\ \mathit{ann}\ x \qquad \text{-- unpack as unboxed basic type (Int, Float, ...)} \\
  & \mid & \textbf{enum}\ x \qquad \text{-- unpack as enum} \\
  & \mid & .. \qquad \text{-- etc.} \\
\mathit{ann} & ::= & .. \qquad \text{-- info about size, ...}
\end{array}
$$

Local mutual recursiveness

- Requires special nodes not yet filled with data

Pipeline: finally, a program!

- Different backends tap at different points from the pipeline
- For example, interpreter based backend taps non-whole program analysed Grin:

```
  -- bytecode
static GB_Byte vb1_bytecode [] =
{ /*0 : l1ts08 0x08 */  /* lbldef [.cod 0] */
                        /* funstart [vb1.len] */
                        /* iduse [vb1.x1__1 4 word LoadSrc_TOS_Rel {ldsrcOff = 2, ldsrcNrWords = 1}] */
                        /* [LoadSrc_TOS_Rel {ldsrcOff = 2, ldsrcNrWords = 1}] */
  0x20, 0x08
, /*2 : evalt */
  0xe0, 0x00, 0x00, 0x00, 0x00
, /*7 : l1ts08 0x00 */  /* stackoff [1] */
                        /* iduse [vb1.4 _42_0 !__ 3 _0 annotdflt LoadSrc_TOS] */
  0x20, 0x00
, /*9 : lnt */
  0xfc
, /*10 : casecall */
  0xf6
```

Tree-oriented programming

How is UHC programmed?

- Tree representation, transformation, generation: UU Attribute Grammar system

Tree-oriented programming

How is UHC programmed?

- Tree representation, transformation, generation: UU Attribute Grammar system

But also

- Parser: UU parsing library
- Logistics: *Shuffle* for generating different compilers for different variants & aspects
- Type system: *Ruler* for describing type system & and generating implementation (current research)

Tree-oriented programming

```
data Expr
  =
      Con Int
    | Add Expr Expr
    | Mul Expr Expr
```

Tree-oriented programming

$$
\begin{array}{ll}
\textbf{data}\ \mathit{Expr} & \\
\quad = & \\
\qquad \mathit{Con}\ \mathit{Int} & \\
\quad \mid\ \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} & \\
\quad \mid\ \mathit{Mul}\ \mathit{Expr}\ \mathit{Expr} &
\end{array}
$$

$$\mathit{calc} :: \mathit{Expr} \rightarrow \mathit{Int}$$

Tree-oriented programming

$$
\begin{array}{l}
\textbf{data}\ \mathit{Expr} \\
\quad = \\
\qquad \mathit{Con}\ \ \mathit{Int} \\
\quad \mid\ \ \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \\
\quad \mid\ \ \mathit{Mul}\ \mathit{Expr}\ \mathit{Expr}
\end{array}
$$

$$
\begin{array}{l}
\mathit{fold} \\
\quad :: \\
\qquad (\mathit{Int} \qquad\ \to b) \\
\quad \to (b \to b \to b) \\
\quad \to (b \to b \to b) \\
\quad \to \mathit{Expr} \qquad \to b
\end{array}
$$

$$\mathit{calc} :: \mathit{Expr} \to \mathit{Int}$$

Tree-oriented programming

$$\textbf{data}\ \mathit{Expr} = \mathit{Con}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{Mul}\ \mathit{Expr}\ \mathit{Expr}$$

$$\begin{aligned} \mathit{fold} ::\ & (\mathit{Int} \rightarrow b) \\ \rightarrow\ & (b \rightarrow b \rightarrow b) \\ \rightarrow\ & (b \rightarrow b \rightarrow b) \\ \rightarrow\ & \mathit{Expr} \rightarrow b \end{aligned}$$

$$\mathit{calc} :: \mathit{Expr} \rightarrow \mathit{Int}$$

$$\mathit{calc} = \mathit{fold}\ \mathit{id}\ (+)\ (*)$$

Tree-oriented programming

$$
\begin{array}{l}
\textbf{data}\ \mathit{Expr} \\
\quad = \\
\qquad \mathit{Con}\ \mathit{Int} \\
\quad \mid\ \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \\
\quad \mid\ \mathit{Mul}\ \mathit{Expr}\ \mathit{Expr}
\end{array}
$$

$$
\begin{array}{l}
\mathit{fold} \\
\quad :: \\
\qquad (\mathit{Int} \qquad\quad \rightarrow b) \\
\quad \rightarrow (b \rightarrow b \rightarrow b) \\
\quad \rightarrow (b \rightarrow b \rightarrow b) \\
\quad \rightarrow \mathit{Expr} \qquad \rightarrow b
\end{array}
$$

$$
\begin{array}{l}
\mathit{calc} :: \mathit{Expr} \rightarrow \mathit{Int} \\
\mathit{calc} = \mathit{fold} \\
(\lambda n \quad \rightarrow n \qquad) \\
(\lambda x\ y \rightarrow x + y) \\
(\lambda x\ y \rightarrow x * y\ )
\end{array}
$$

Tree-oriented programming

$$\begin{aligned}&\textbf{data}\ \mathit{Expr}\\ &\ =\\ &\qquad \mathit{Con}\ \mathit{Int}\\ &\ \mid\ \ \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}\\ &\ \mid\ \ \mathit{Mul}\ \mathit{Expr}\ \mathit{Expr}\end{aligned}$$

$$\begin{aligned}&\textbf{type}\ \mathit{Sem}\ b\\ &\ =\\ &\ (\ (\mathit{Int} \rightarrow b)\\ &\ ,\ (b \rightarrow b \rightarrow b)\\ &\ ,\ (b \rightarrow b \rightarrow b)\\ &\ )\end{aligned}$$

$$\begin{aligned}&\mathit{fold} :: \mathit{Sem}\ b \rightarrow\\ &\qquad\quad \mathit{Expr} \rightarrow b\end{aligned}$$

$$\begin{aligned}&\mathit{calc} :: \mathit{Expr} \rightarrow \mathit{Int}\\ &\mathit{calc} = \mathit{fold}\\ &(\lambda n \rightarrow n\ )\\ &(\lambda x\ y \rightarrow x + y)\\ &(\lambda x\ y \rightarrow x * y\ )\end{aligned}$$

Tree-oriented programming

$$
\begin{aligned}
&\textbf{data}\ \mathit{Expr} \\
&\quad = \\
&\qquad \mathit{Con}\ \mathit{Int} \\
&\quad \mid\ \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \\
&\quad \mid\ \mathit{Mul}\ \mathit{Expr}\ \mathit{Expr}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{type}\ \mathit{Sem}\ b \\
&\ = \\
&\ (\ (\mathit{Int} \qquad \rightarrow b) \\
&\ ,\ \ (b \rightarrow b \rightarrow b) \\
&\ ,\ \ (b \rightarrow b \rightarrow b) \\
&\ )
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{calcsem} :: \mathit{Sem}\ \mathit{Int} \\
&\mathit{calcsem} = \\
&(\lambda n \quad \rightarrow n \\
&,\lambda x\ y \rightarrow x + y \\
&,\lambda x\ y \rightarrow x * y \\
&)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{fold} :: \mathit{Sem}\ b \rightarrow \\
&\qquad\quad \mathit{Expr} \rightarrow b
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{calc}\ :: \mathit{Expr} \rightarrow \mathit{Int} \\
&\mathit{calc} = \mathit{fold}\ \mathit{calcsem}
\end{aligned}
$$

Tree-oriented programming

$$
\begin{array}{l}
\textbf{data}\ \mathit{Expr} \\
\quad = \\
\qquad \mathit{Con}\ \mathit{Int} \\
\quad \mid\ \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \\
\quad \mid\ \mathit{Mul}\ \mathit{Expr}\ \mathit{Expr} \\
\quad \mid\ \mathit{Var}\ \mathit{Name}
\end{array}
$$

$$
\begin{array}{l}
\textbf{type}\ \mathit{Sem}\ b \\
\quad = \\
\quad (\ (\mathit{Int} \rightarrow b) \\
\quad ,\ (b \rightarrow b \rightarrow b) \\
\quad ,\ (b \rightarrow b \rightarrow b) \\
\quad)
\end{array}
$$

$$
\begin{array}{l}
\mathit{fold} :: \mathit{Sem}\ b \rightarrow \\
\qquad\qquad \mathit{Expr} \rightarrow b
\end{array}
$$

$$
\begin{array}{l}
\mathit{calcsem} :: \mathit{Sem}\ \mathit{Int} \\
\mathit{calcsem} = \\
(\lambda n \rightarrow n \\
,\lambda x\ y \rightarrow x + y \\
,\lambda x\ y \rightarrow x * y \\
)
\end{array}
$$

$$
\begin{array}{l}
\mathit{calc}\ :: \mathit{Expr} \rightarrow \mathit{Int} \\
\mathit{calc} = \mathit{fold}\ \mathit{calcsem}
\end{array}
$$

Tree-oriented programming

$$\begin{array}{l} \textbf{data}\ \mathit{Expr} \\ \quad = \\ \qquad \mathit{Con}\ \mathit{Int} \\ \quad \mid\ \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \\ \quad \mid\ \mathit{Mul}\ \mathit{Expr}\ \mathit{Expr} \\ \quad \mid\ \mathit{Var}\ \mathit{Name} \end{array}$$

$$\begin{array}{l} \textbf{type}\ \mathit{Sem}\ b \\ \quad = \\ \quad (\ (\mathit{Int} \rightarrow b) \\ \quad ,\ (b \rightarrow b \rightarrow b) \\ \quad ,\ (b \rightarrow b \rightarrow b) \\ \quad ,\ (\mathit{Name} \rightarrow b) \\ \quad) \end{array}$$

$$\begin{array}{l} \mathit{fold} :: \mathit{Sem}\ b \rightarrow \\ \qquad\qquad \mathit{Expr} \rightarrow b \end{array}$$

$$\begin{array}{l} \mathit{calcsem} :: \mathit{Sem}\ \mathit{Int} \\ \mathit{calcsem} = \\ (\lambda n \rightarrow n \\ ,\lambda x\ y \rightarrow x + y \\ ,\lambda x\ y \rightarrow x * y \\ ) \end{array}$$

$$\begin{array}{l} \mathit{calc} :: \mathit{Expr} \rightarrow \mathit{Int} \\ \mathit{calc} = \mathit{fold}\ \mathit{calcsem} \end{array}$$

Tree-oriented programming

$$\begin{array}{ll} \textbf{data}\ \mathit{Expr} & \\ \quad = & \\ \quad\quad & \mathit{Con}\ \mathit{Int} \\ \quad\mid & \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \\ \quad\mid & \mathit{Mul}\ \mathit{Expr}\ \mathit{Expr} \\ \quad\mid & \mathit{Var}\ \ \mathit{Name} \end{array}$$

$$\begin{array}{l} \textbf{type}\ \mathit{Sem}\ b \\ = \\ (\ \ (\mathit{Int} \quad\quad \to b) \\ ,\ \ (b \to b \to b) \\ ,\ \ (b \to b \to b) \\ ,\ \ (\mathit{Name} \to b) \\ ) \end{array}$$

$$\begin{array}{l} \mathit{fold} :: \mathit{Sem}\ b \to \\ \quad\quad\quad\quad \mathit{Expr} \to b \end{array}$$

$$\begin{array}{l} \mathit{calcsem} :: \mathit{Sem}\ \mathit{Int} \\ \mathit{calcsem} = \\ (\lambda n \quad\ \to n \\ ,\lambda x\ y \to x + y \\ ,\lambda x\ y \to x * y \\ ,\lambda s \quad\ \to \mathit{lookup}\ s\ e \\ ) \end{array}$$

$$\begin{array}{l} \mathit{calc}\ :: \mathit{Expr} \to \mathit{Int} \\ \mathit{calc} = \mathit{fold}\ \mathit{calcsem} \end{array}$$

Tree-oriented programming

$$\begin{array}{l} \textbf{data}\ \mathit{Expr} \\ \quad = \\ \qquad \mathit{Con}\ \mathit{Int} \\ \quad \mid\ \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \\ \quad \mid\ \mathit{Mul}\ \mathit{Expr}\ \mathit{Expr} \\ \quad \mid\ \mathit{Var}\ \mathit{Name} \end{array}$$

$$\begin{array}{l} \textbf{type}\ \mathit{Sem}\ b \\ \quad = \\ \quad (\ (\mathit{Int} \to b) \\ \quad ,\ (b \to b \to b) \\ \quad ,\ (b \to b \to b) \\ \quad ,\ (\mathit{Name} \to b) \\ \quad) \end{array}$$

$$\begin{array}{l} \mathit{fold} :: \mathit{Sem}\ b \to \\ \qquad \mathit{Expr} \to b \end{array}$$

$$\begin{array}{l} \mathit{calcsem} :: \mathit{Sem}\ \mathit{Int} \\ \mathit{calcsem} = \\ (\lambda n \to n \\ , \lambda x\ y \to x + y \\ , \lambda x\ y \to x * y \\ , \lambda s \to \lambda e \to \mathit{lookup}\ s\ e \\ ) \end{array}$$

$$\begin{array}{l} \mathit{calc} :: \mathit{Expr} \to \mathit{Int} \\ \mathit{calc} = \mathit{fold}\ \mathit{calcsem} \end{array}$$

Tree-oriented programming

$$
\begin{array}{l}
\textbf{data}\ \mathit{Expr} \\
\quad = \\
\qquad \mathit{Con}\ \ \mathit{Int} \\
\quad \mid\ \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \\
\quad \mid\ \mathit{Mul}\ \mathit{Expr}\ \mathit{Expr} \\
\quad \mid\ \mathit{Var}\ \ \mathit{Name}
\end{array}
$$

$$
\begin{array}{l}
\textbf{type}\ \mathit{Sem}\ b \\
\quad = \\
\quad (\ (\mathit{Int} \rightarrow b) \\
\quad ,\ (b \rightarrow b \rightarrow b) \\
\quad ,\ (b \rightarrow b \rightarrow b) \\
\quad ,\ (\mathit{Name} \rightarrow b) \\
\quad)
\end{array}
$$

$$
\begin{array}{l}
\mathit{fold} :: \mathit{Sem}\ b \rightarrow \\
\qquad \mathit{Expr} \rightarrow b
\end{array}
$$

$$
\begin{array}{l}
\mathit{calcsem} :: \mathit{Sem}\ (\mathit{Env} \rightarrow \mathit{Int}) \\
\mathit{calcsem} = \\
(\lambda n \rightarrow n \\
,\lambda x\ y \rightarrow x + y \\
,\lambda x\ y \rightarrow x * y \\
,\lambda s \rightarrow \lambda e \rightarrow \mathit{lookup}\ s\ e \\
)
\end{array}
$$

$$
\begin{array}{l}
\mathit{calc} :: \mathit{Expr} \rightarrow \mathit{Int} \\
\mathit{calc} = \mathit{fold}\ \mathit{calcsem}
\end{array}
$$

Tree-oriented programming

$$
\begin{aligned}
&\textbf{data}\ Expr\\
&\quad =\\
&\qquad Con\ Int\\
&\quad\mid\ Add\ Expr\ Expr\\
&\quad\mid\ Mul\ Expr\ Expr\\
&\quad\mid\ Var\ \ Name
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{type}\ Sem\ b\\
&\ =\\
&\ (\ (Int \to b)\\
&\ ,\ (b \to b \to b)\\
&\ ,\ (b \to b \to b)\\
&\ ,\ (Name \to b)\\
&\ )
\end{aligned}
$$

$$
\begin{aligned}
&fold :: Sem\ b \to\\
&\qquad\quad Expr \to b
\end{aligned}
$$

$$
\begin{aligned}
&calcsem :: Sem\ (Env \to Int)\\
&calcsem =\\
&(\lambda n \to \lambda e \to n\\
&,\lambda x\ y \to \lambda e \to x\ e + y\ e\\
&,\lambda x\ y \to \lambda e \to x\ e * y\ e\\
&,\lambda s \to \lambda e \to lookup\ s\ e\\
&)
\end{aligned}
$$

$$
\begin{aligned}
&calc :: Expr \to Int\\
&calc = fold\ calcsem\ testenv
\end{aligned}
$$

Tree-oriented programming

$$
\begin{array}{l}
\textbf{data}\ \mathit{Expr} \\
\quad = \\
\qquad \mathit{Con}\ \mathit{Int} \\
\quad \mid\ \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \\
\quad \mid\ \mathit{Mul}\ \mathit{Expr}\ \mathit{Expr} \\
\quad \mid\ \mathit{Var}\ \mathit{Name}
\end{array}
$$

$$
\begin{array}{l}
\textbf{type}\ \mathit{Sem}\ b \\
= \\
(\ (\mathit{Int} \rightarrow b) \\
,\ (b \rightarrow b \rightarrow b) \\
,\ (b \rightarrow b \rightarrow b) \\
,\ (\mathit{Name} \rightarrow b) \\
) \\
\\
\mathit{fold} :: \mathit{Sem}\ b \rightarrow \\
\qquad \mathit{Expr} \rightarrow b
\end{array}
$$

$$
\begin{array}{l}
\mathit{calcsem} :: \mathit{Sem}\ (\mathit{Env} \rightarrow \mathit{Int}) \\
\mathit{calcsem} = \\
(\lambda n \rightarrow \lambda e \rightarrow n \\
,\lambda x\ y \rightarrow \lambda e \rightarrow x\ e + y\ e \\
,\boxed{\lambda x\ y \rightarrow \lambda e \rightarrow x\ e * y\ e} \\
,\lambda s \rightarrow \lambda e \rightarrow \mathit{lookup}\ s\ e \\
) \\
\\
\mathit{calc} :: \mathit{Expr} \rightarrow \mathit{Int} \\
\mathit{calc} = \mathit{fold}\ \mathit{calcsem}\ \mathit{testenv}
\end{array}
$$

Tree-oriented programming

```
data Expr
  =
    Con Int
  | Add Expr Expr
  | Mul Expr Expr
  | Var Name
```

Fields

```
type Sem b
  =
  ( (Int          → b)
  , (b → b → b)
  , (b → b → b)
  , (Name → b)
  )

fold :: Sem b →
           Expr → b
```

```
calcsem :: Sem (Env → Int)
ca
( λ        e → n
, λx y → λe → x e + y e
, λx y → λe → x e ∗ y e
, λs    → λe → lookup s e
)

calc :: Expr → Int
calc = fold calcsem testenv
```

Inherited attribute

Synthesized attribute

Tree-oriented programming

data *Expr*
$=$
$\quad Con\ Int$
$\mid Add\ Expr\ Expr$
$\mid Mul\ Expr\ Expr$
$\mid Var\ \ Name$

Fields

type *Sem b*
$=$
$(\ (Int \rightarrow b)$
$,\ (b \rightarrow b \rightarrow b)$
$,\ (b \rightarrow b \rightarrow b)$
$,\ (Name \rightarrow b)$
$)$

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

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

Inherited attribute

Synthesized attribute

ca
$(\lambda$ … $e \rightarrow i$
$, \lambda x\ y \rightarrow \lambda e \rightarrow x\ e + y\ e$
$, \lambda x\ y \rightarrow \lambda e \rightarrow x\ e * y\ e$
$, \lambda s \rightarrow \lambda e \rightarrow lookup\ s\ e$
$)$

$calc :: Expr \rightarrow Int$
$calc = fold\ calcsem\ testenv$

Tree-oriented programming

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Expr} & & \\
\ = & & \\
 & \mathit{Con} & \mathit{con} : \mathit{Int} \\
 | & \mathit{Add} & \mathit{lef}\ : \mathit{Expr}\ \ \mathit{rit} : \mathit{Expr} \\
 | & \mathit{Mul} & \mathit{lef}\ : \mathit{Expr}\ \ \mathit{rit} : \mathit{Expr} \\
 | & \mathit{Var} & \mathit{name} : \mathit{Name}
\end{array}
$$

Named fields

$$
\begin{array}{l}
\mathit{calcsem} :: \mathit{Sem}\ (\mathit{Env} \rightarrow \mathit{Int}) \\
\mathit{ca} \\
(\ldots e \rightarrow n \\
, \lambda x\ y \rightarrow \lambda e \rightarrow x \\
, \boxed{\lambda x\ y \rightarrow \lambda e \rightarrow x\ e * y\ e} \\
, \lambda s \quad\ \rightarrow \lambda e \rightarrow \mathit{lookup}\ s\ e \\
)
\end{array}
$$

Inherited attribute

Synthesized attribute

Tree-oriented programming

data *Expr*

$$
\begin{aligned}
= \; & \mathit{Con} \;\; \mathit{con} : \mathit{Int} \\
\mid \; & \mathit{Add} \;\; \mathit{lef} : \mathit{Expr} \;\; \mathit{rit} : \mathit{Expr} \\
\mid \; & \mathit{Mul} \;\; \mathit{lef} : \mathit{Expr} \;\; \mathit{rit} : \mathit{Expr} \\
\mid \; & \mathit{Var} \;\; \mathit{name} : \mathit{Name}
\end{aligned}
$$

Named fields

$\mathit{calcsem} :: \mathit{Sem}\ (\mathit{Env} \to \mathit{Int})$

ca

$(\lambda \quad e \to n$

$, \lambda x\ y \to \lambda e \to x\ e + y\ e$

$, \lambda x\ y \to \lambda e \to x\ e * y\ e$

$, \lambda s \quad \to \lambda e \to \mathit{lookup}\ s\ e$

$)$

Inherited attribute

Synthesized attribute

Named attributes

attr *Expr* **inh** *env* : *Env*
syn *val* : *Int*

Tree-oriented programming

$$
\begin{aligned}
&\textbf{data}\ \mathit{Expr}\\
&\quad =\\
&\qquad \mathit{Con}\ \ \mathit{con} : \mathit{Int}\\
&\quad \mid\ \mathit{Add}\ \ \mathit{lef} : \mathit{Expr}\ \ \mathit{rit} : \mathit{Expr}\\
&\quad \mid\ \mathit{Mul}\ \ \mathit{lef} : \mathit{Expr}\ \ \mathit{rit} : \mathit{Expr}\\
&\quad \mid\ \mathit{Var}\ \ \mathit{name} : \mathit{Name}
\end{aligned}
$$

Named fields

$$
\begin{aligned}
&\mathit{calcsem} :: \mathit{Sem}\ (\mathit{Env} \to \mathit{Int})\\
&\mathit{c}\\
&(\ \ldots \to \mathit{r}\\
&,\lambda x\ y \to \lambda e \to x\ \ldots\\
&,\boxed{\lambda x\ y \to \lambda e \to x\ e * y\ e}\\
&,\lambda s\ \ \ \ \to \lambda e \to \mathit{lookup}\ s\ e\\
&)
\end{aligned}
$$

Inherited attribute

Synthesized attribute

Named attributes

$$
\begin{aligned}
&\textbf{attr}\ \mathit{Expr}\ \textbf{inh}\ \mathit{env} : \mathit{Env}\\
&\qquad\qquad\quad \textbf{syn}\ \mathit{val} : \mathit{Int}
\end{aligned}
$$

$$
\begin{aligned}
\textbf{sem}\ \mathit{Expr} \mid \mathit{Mul}\ \textbf{lhs}.\mathit{val} &= @\mathit{lef}.\mathit{val} * @\mathit{rit}.\mathit{val}\\
\mathit{lef}.\mathit{env} &= @\textbf{lhs}.\mathit{env}\\
\mathit{rit}.\mathit{env} &= @\textbf{lhs}.\mathit{env}
\end{aligned}
$$

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:

$$\begin{array}{l} \textbf{let}\ y = z\ \textbf{in} \\ \textbf{let}\ x = y\ \textbf{in} \\ \quad x + y \end{array} \quad \longrightarrow \quad \begin{array}{l} \textbf{let}\ y = z\ \textbf{in} \\ \textbf{let}\ x = y\ \textbf{in} \\ \quad {\color{red}z} + {\color{red}z} \end{array}$$

Inline name aliases

AG applied in UHC

Desired Core transformation:

$$\begin{array}{lcl} \textbf{let } y = z \textbf{ in} & & \textbf{let } y = z \textbf{ in} \\ \textbf{let } x = y \textbf{ in} & \longrightarrow & \textbf{let } x = y \textbf{ in} \\ \quad x + y & & \quad z + z \end{array}$$

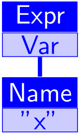
Inline name aliases

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

Example Core transformation

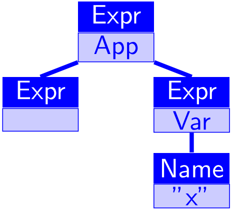
$\mathbf{data}\ \mathit{Expr} = \mathit{Var}\ \ \mathit{nm} : \mathit{Name}$

Example Core transformation



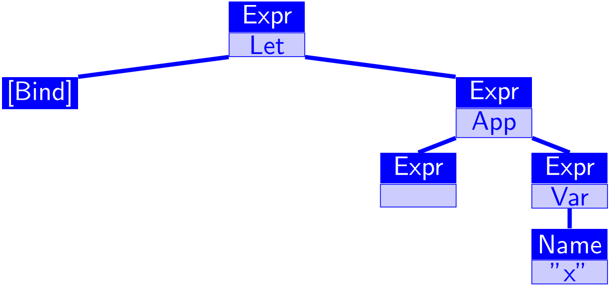
data *Expr* = *Var* *nm* : *Name*

Example Core transformation



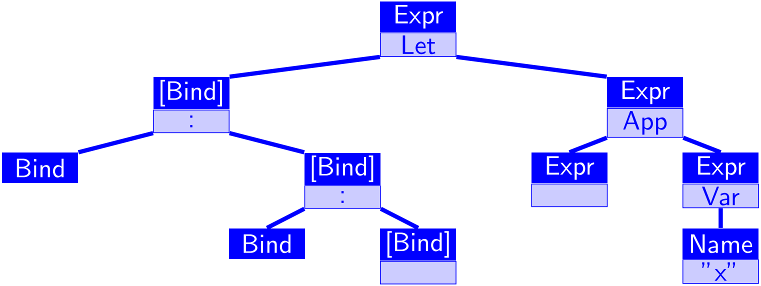
$$\textbf{data}\ \mathit{Expr} = \mathit{Var}\ \mathit{nm} : \mathit{Name}$$
$$\quad\quad\quad\quad\ \mid\ \mathit{App}\ f : \mathit{Expr}\ a : \mathit{Expr}$$

Example Core transformation



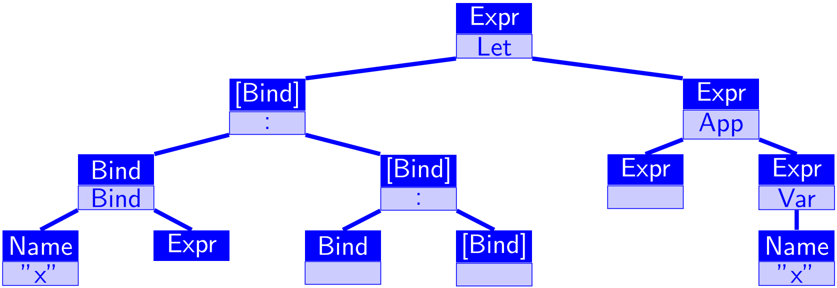
$$
\begin{aligned}
\textbf{data}\ \mathit{Expr} &= \mathit{Var}\ \mathit{nm} : \mathit{Name} \\
&\mid\ \mathit{App}\ f : \mathit{Expr}\ \ a : \mathit{Expr} \\
&\mid\ \mathit{Let}\ \mathit{defs} : [\mathit{Bind}]\ \mathit{body} : \mathit{Expr}
\end{aligned}
$$

Example Core transformation



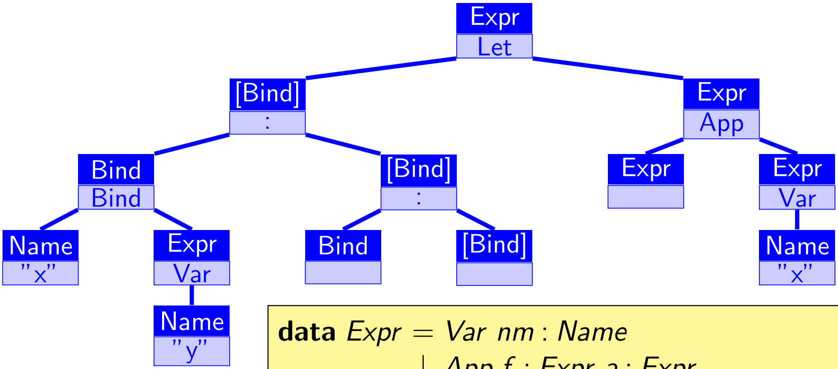
$$\begin{aligned}\textbf{data}\ \mathit{Expr} = {} & \mathit{Var}\ \mathit{nm} : \mathit{Name} \\ \mid {} & \mathit{App}\ \mathit{f} : \mathit{Expr}\ \mathit{a} : \mathit{Expr} \\ \mid {} & \mathit{Let}\ \mathit{defs} : [\mathit{Bind}]\ \mathit{body} : \mathit{Expr}\end{aligned}$$

Example Core transformation



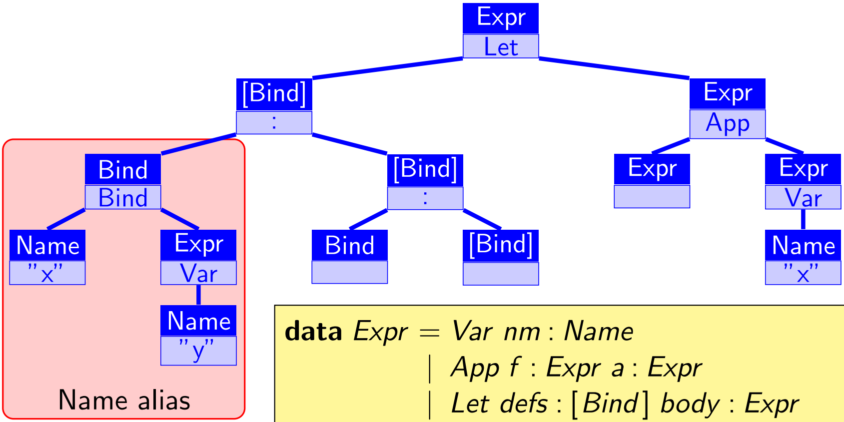
data *Expr* = *Var nm* : *Name*
| *App f* : *Expr a* : *Expr*
| *Let defs* : [*Bind*] *body* : *Expr*
data *Bind* = *Bind Name Expr*

Example Core transformation



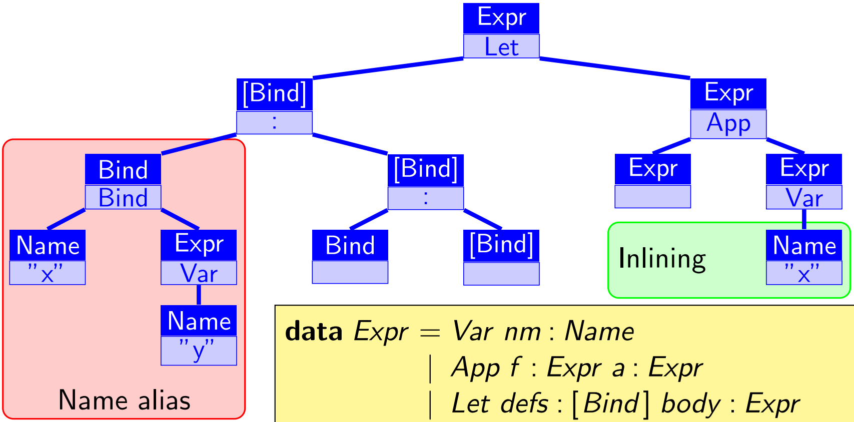
data *Expr* = *Var nm* : *Name*
| *App f* : *Expr a* : *Expr*
| *Let defs* : [*Bind*] *body* : *Expr*
data *Bind* = *Bind Name Expr*

Example Core transformation: Name alias



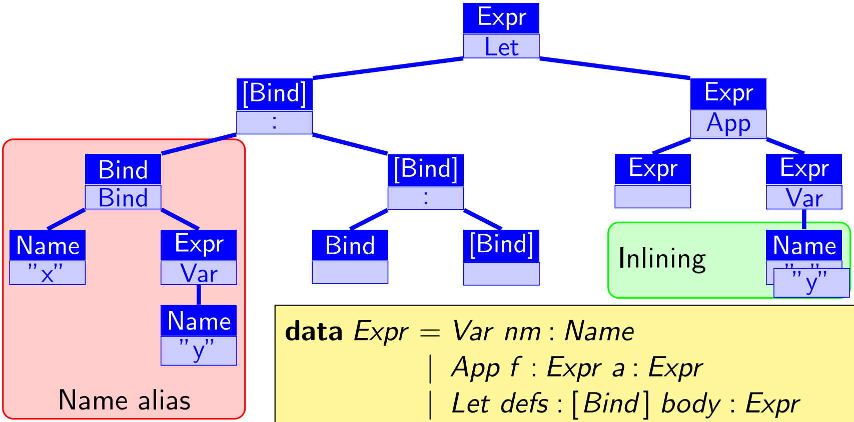

```
data Expr = Var nm : Name
          | App f : Expr a : Expr
          | Let defs : [Bind] body : Expr
data Bind = Bind Name Expr
```

Example Core transformation: Name alias inlining



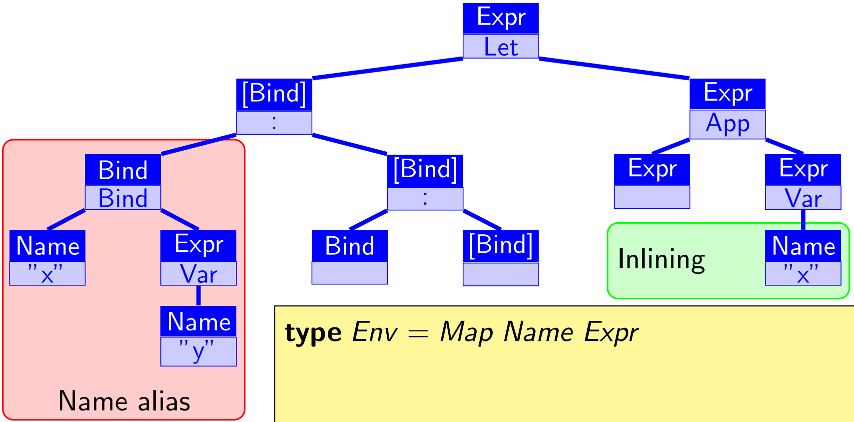
$$
\begin{aligned}
\textbf{data}\ \mathit{Expr} &= \mathit{Var}\ \mathit{nm} : \mathit{Name} \\
&\mid\ \mathit{App}\ f : \mathit{Expr}\ a : \mathit{Expr} \\
&\mid\ \mathit{Let}\ \mathit{defs} : [\mathit{Bind}]\ \mathit{body} : \mathit{Expr} \\
\textbf{data}\ \mathit{Bind} &= \mathit{Bind}\ \mathit{Name}\ \mathit{Expr}
\end{aligned}
$$

Example Core transformation: Name alias inlining



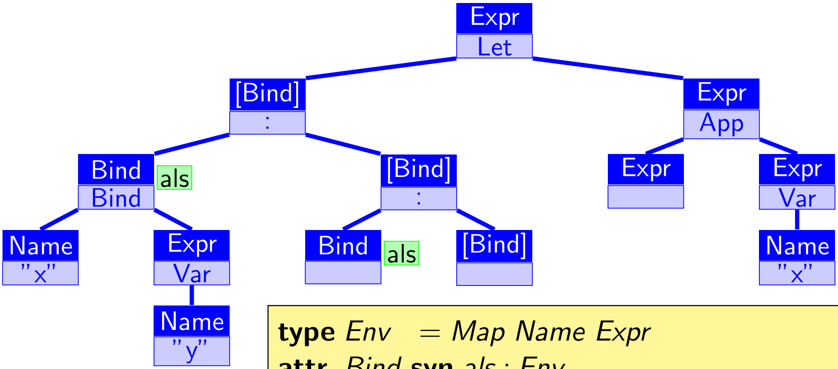
$$
\begin{array}{lll}
\mathbf{data}\ \mathit{Expr} & = & \mathit{Var}\ \mathit{nm} : \mathit{Name} \\
 & \mid & \mathit{App}\ \mathit{f} : \mathit{Expr}\ \mathit{a} : \mathit{Expr} \\
 & \mid & \mathit{Let}\ \mathit{defs} : [\mathit{Bind}]\ \mathit{body} : \mathit{Expr} \\
\mathbf{data}\ \mathit{Bind} & = & \mathit{Bind}\ \mathit{Name}\ \mathit{Expr}
\end{array}
$$

Example Core transformation: Name alias inlining



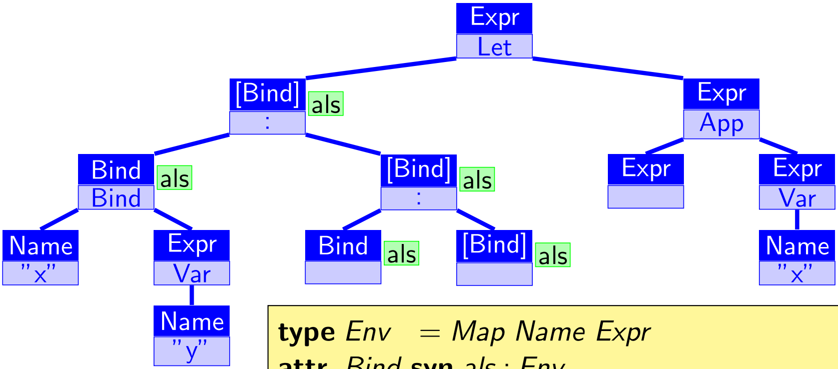
type *Env* = *Map Name Expr*

Example Core transformation: Name alias inlining



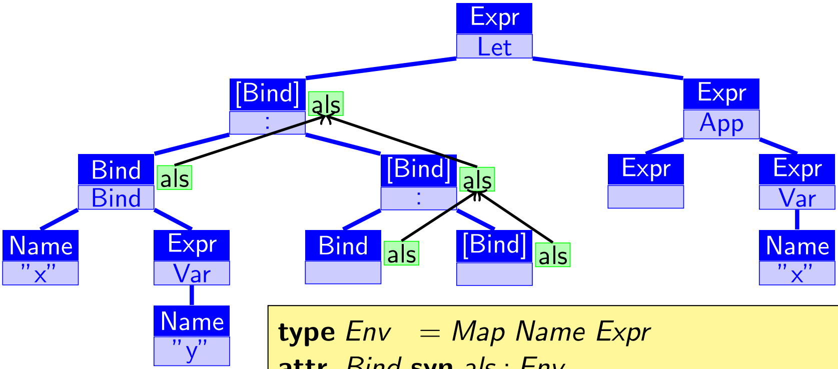
type *Env* = *Map Name Expr*
attr *Bind* **syn** *als* : *Env*

Example Core transformation: Name alias inlining



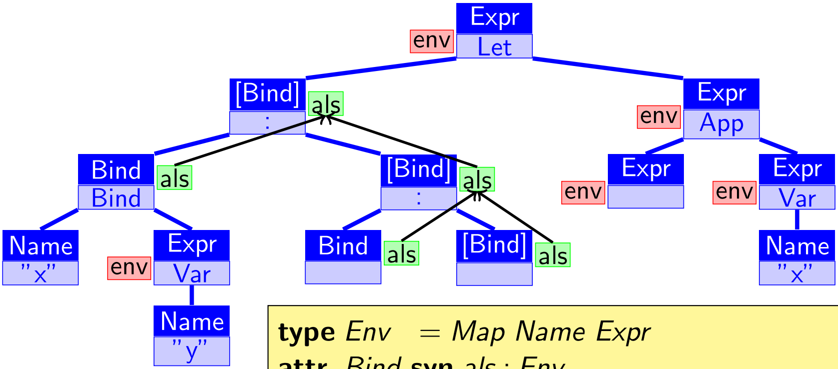
type *Env* = *Map Name Expr*
attr *Bind* **syn** *als* : *Env*
attr [*Bind*] **syn** *als* : *Env*

Example Core transformation: Name alias inlining



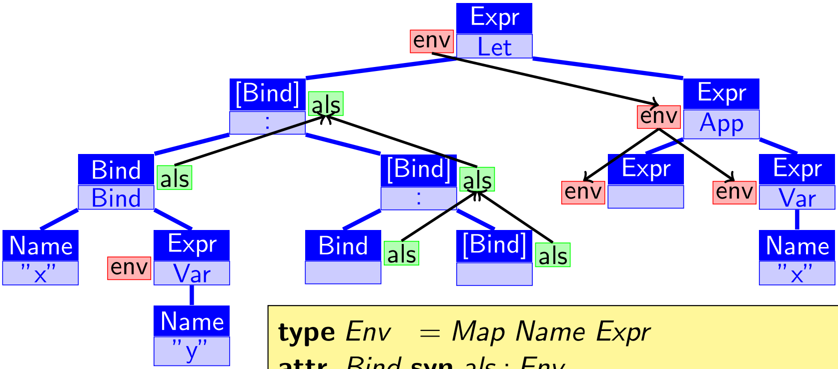
type *Env* = *Map Name Expr*
attr *Bind* **syn** *als* : *Env*
attr [*Bind*] **syn** *als* **use** $\{\cup\}$ $\{\emptyset\}$: *Env*

Example Core transformation: Name alias inlining



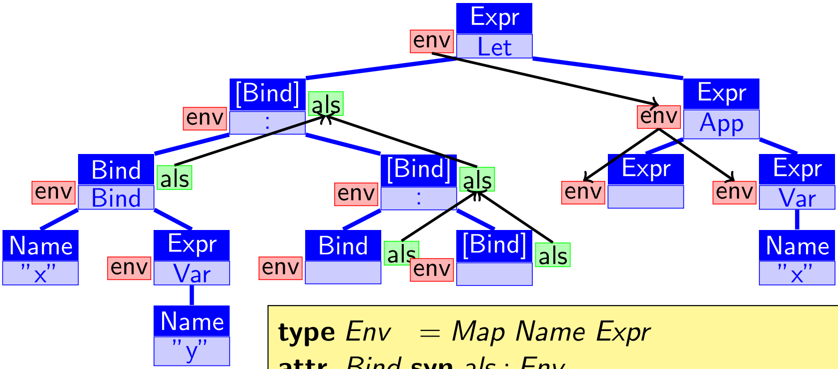
type *Env* $=$ *Map Name Expr*
attr *Bind* **syn** *als* : *Env*
attr [*Bind*] **syn** *als* **use** $\{\cup\}$ $\{\emptyset\}$: *Env*
attr *Expr* **inh** *env* : *Env*

Example Core transformation: Name alias inlining



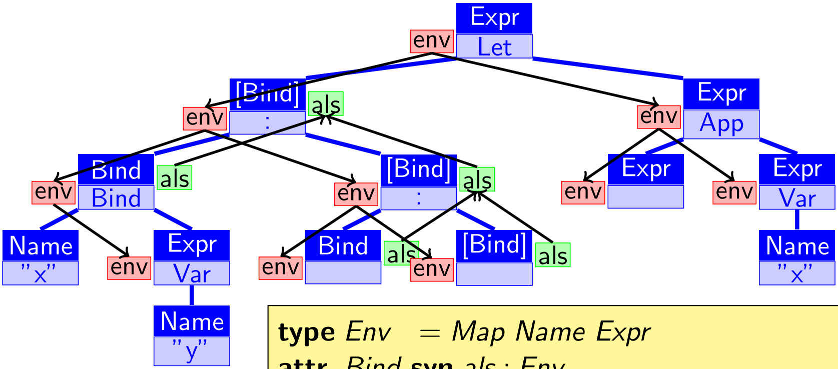
type *Env* = *Map Name Expr*
attr *Bind* **syn** *als* : *Env*
attr [*Bind*] **syn** *als* **use** $\{\cup\}$ $\{\emptyset\}$: *Env*
attr *Expr* **inh** *env* : *Env*

Example Core transformation: Name alias inlining



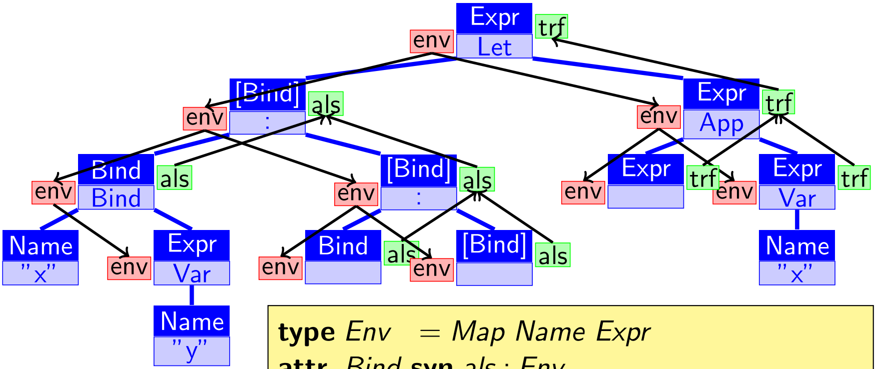
type *Env* $=$ *Map Name Expr*
attr *Bind* **syn** *als* : *Env*
attr [*Bind*] **syn** *als* **use** $\{\cup\}$ $\{\emptyset\}$: *Env*
attr *Expr* **inh** *env* : *Env*
attr *Bind* [*Bind*] **inh** *env* : *Env*

Example Core transformation: Name alias inlining



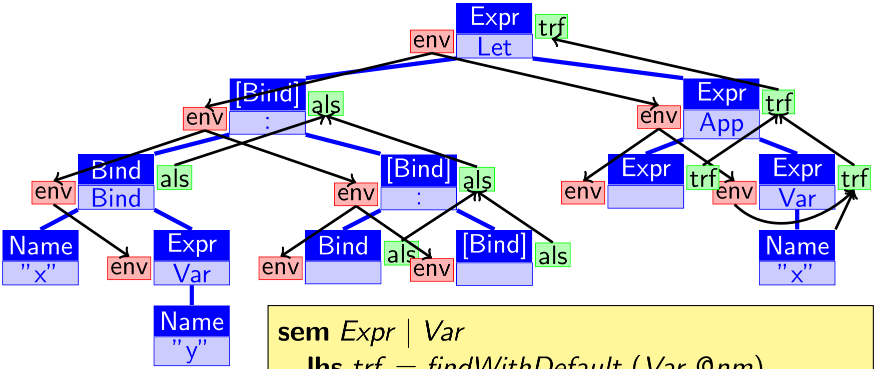
type *Env* = *Map Name Expr*
attr *Bind* **syn** *als* : *Env*
attr [*Bind*] **syn** *als* **use** $\{\cup\}$ $\{\emptyset\}$: *Env*
attr *Expr* **inh** *env* : *Env*
attr *Bind* [*Bind*] **inh** *env* : *Env*

Example Core transformation: Name alias inlining



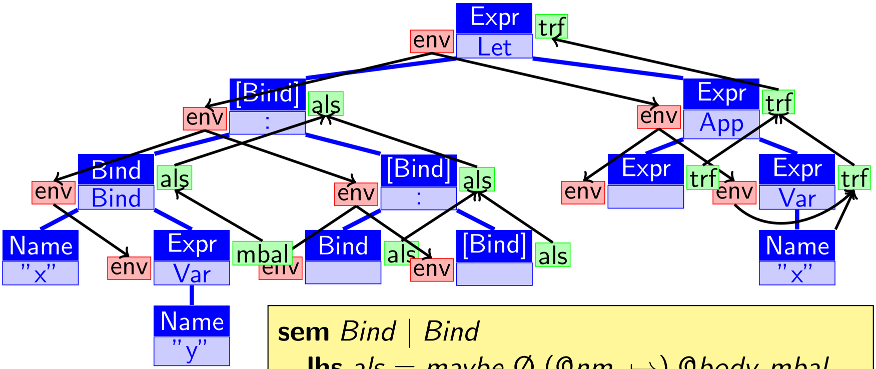
type *Env* $=$ *Map Name Expr*
attr *Bind* **syn** *als* : *Env*
attr [*Bind*] **syn** *als* **use** $\{\cup\}$ $\{\emptyset\}$: *Env*
attr *Expr* **inh** *env* : *Env*
attr *Bind* [*Bind*] **inh** *env* : *Env*
attr *Expr Bind* [*Bind*] **syn** *trf* : **self**

Example Core transformation: Name alias inlining



sem *Expr* | *Var*
  lhs.*trf* = *findWithDefault* (*Var* @*nm*)
                  @*nm*
                  @**lhs**.*env*

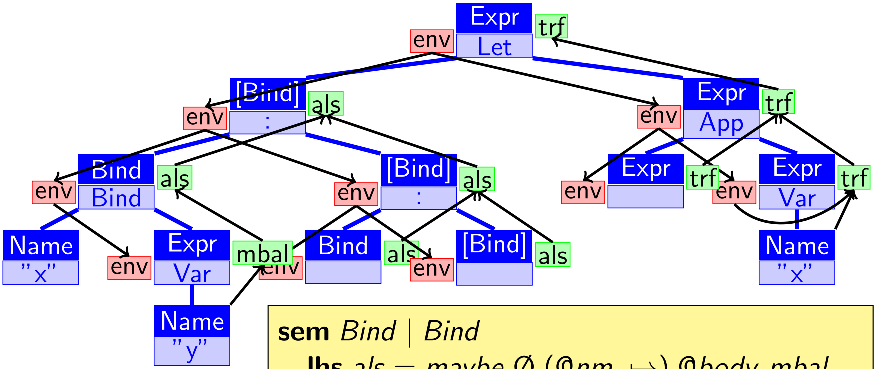
Example Core transformation: Name alias inlining



sem *Bind* | *Bind*

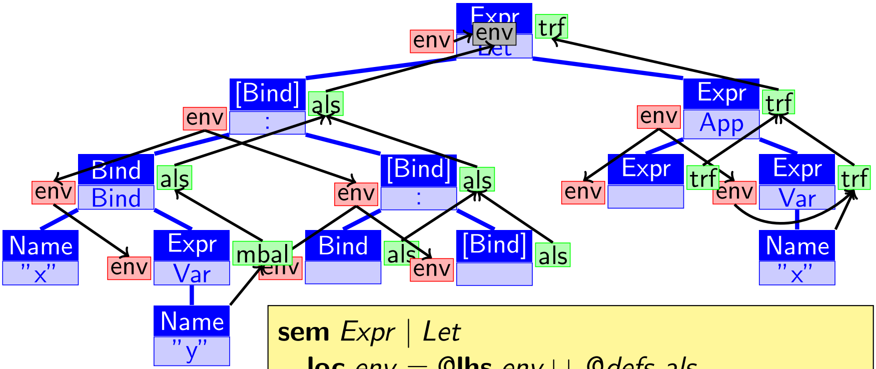
  lhs.*als* = *maybe* ∅ (@*nm* ↦) @*body*.*mbal*

Example Core transformation: Name alias inlining




```
sem Bind | Bind
    lhs.als = maybe ∅ (@nm ↦) @body.mbal
attr Expr syn mbal : Maybe Expr
sem Expr | Var
    lhs.mbal = Just (Var @nm)
```

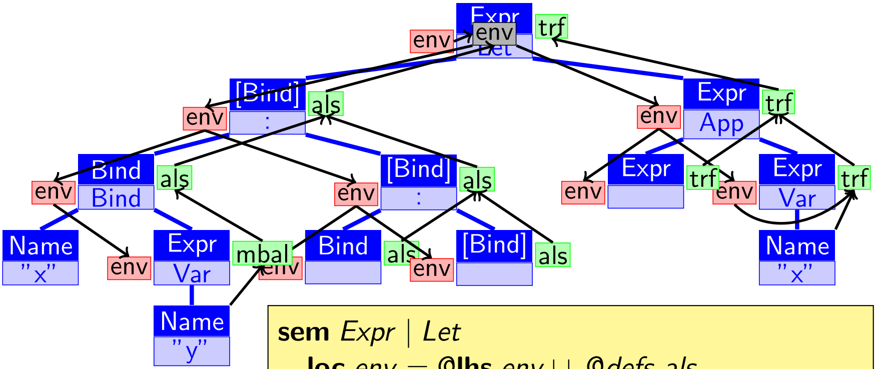
Example Core transformation: Name alias inlining



sem *Expr* | *Let*

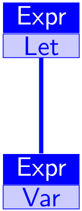
  loc.*env* = @**lhs**.*env* ∪ @*defs*.*als*

Example Core transformation: Name alias inlining



sem *Expr* | *Let*
    loc.*env* = @**lhs**.*env* ∪ @*defs*.*als*

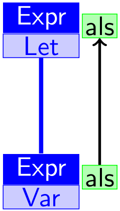
Multi-pass tree traversal



Name alias inlining is a two-pass traversal:

- First pass
- Second pass

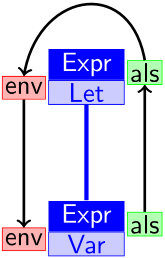
Multi-pass tree traversal



Name alias inlining is a two-pass traversal:

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

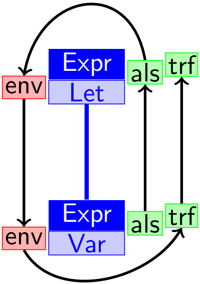
Multi-pass tree traversal



Name alias inlining is a two-pass traversal:

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

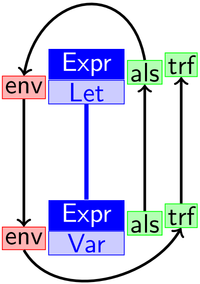
Multi-pass tree traversal



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*

Multi-pass tree traversal



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)

Project status

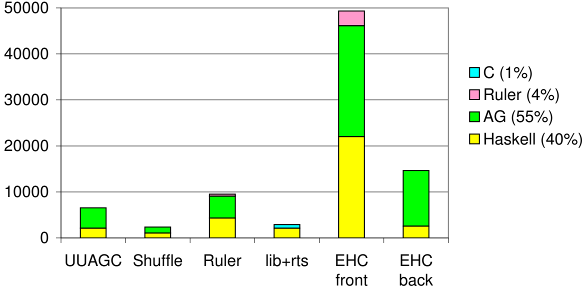
Status of the Essential Haskell Compiler

- Available on `www.cs.uu.nl/wiki/Ehc`

Project status

Status of the Essential Haskell Compiler

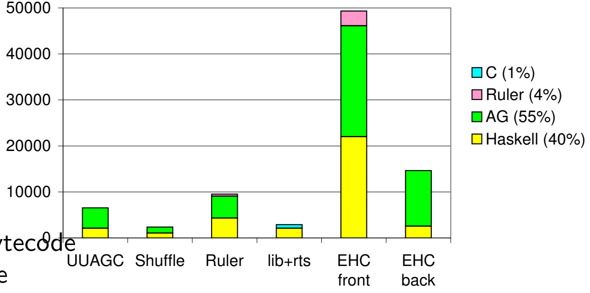
- Available on `www.cs.uu.nl/wiki/Ehc`
- 85000 lines of code,
  half of which in AG



Project status

Status of the Essential Haskell Compiler

- Available on `www.cs.uu.nl/wiki/Ehc`
- 85000 lines of code,
  half of which in AG
- Working towards
  full Haskell
  with full prelude
- Simple programs
  compile and run
  - as interpreted bytecode
  - as compiled code



Summary

Coping with Compiler Complexity

in the Essential Haskell Compiler

Summary

Coping with Compiler Complexity

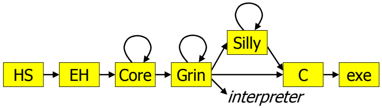
- Implementation complexity
- Description complexity
- Design complexity
- Maintenance complexity

in the Essential Haskell Compiler

Summary

Coping with Compiler Complexity

- Implementation complexity
  Transform!
- Description complexity
- Design complexity
- Maintenance complexity

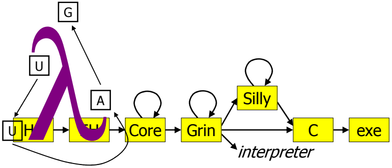


in the Essential Haskell Compiler

Summary

Coping with Compiler Complexity

- Implementation complexity
  Transform!
- Description complexity
  Use tools!
- Design complexity
- Maintenance complexity



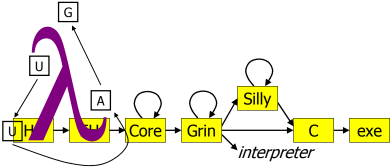
in the Essential Haskell Compiler

Atze Dijkstra, Jeroen Fokker, Doaitse Swierstra

Summary

Coping with Compiler Complexity

- Implementation complexity
  Transform!
- Description complexity
  Use tools!
- Design complexity
  Grow stepwise!
- Maintenance complexity



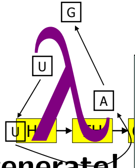
in the Essential Haskell Compiler

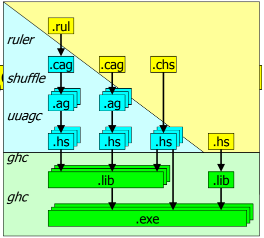
Summary

Coping with Compiler Complexity

- Implementation complexity
  Transform!
- Description complexity
  Use tools!
- Design complexity
  Grow stepwise!
- Maintenance complexity
  Generate, generate, generate!

in the Essential Haskell Compiler

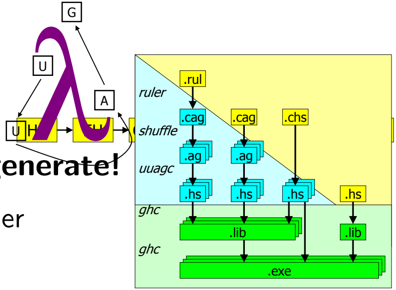




Summary

Coping with Compiler Complexity

- Implementation complexity
  Transform!
- Description complexity
  Use tools!
- Design complexity
  Grow stepwise!
- Maintenance complexity
  Generate, generate, generate!



in the Essential Haskell Compiler

`www.cs.uu.nl/wiki/Ehc`