Essential Haskell (EH) project overview

Atze Dijkstra

September 8, 2006

Why and what?

- EHC (Essential Haskell compiler) is a compiler
  - For Haskell restricted to its essentials
  - Each language feature as general as possible
  - For experimentation and education
- The implementation of the compiler
  - Is partitioned into steps, building on top of each other
  - Each step adds a language feature
  - Each step implements a working compiler which can be used as starting point for changes
- Design starting point
  - Use explicit (type) information provided by programmer
  - And make best effort to propagate this information to where it is needed

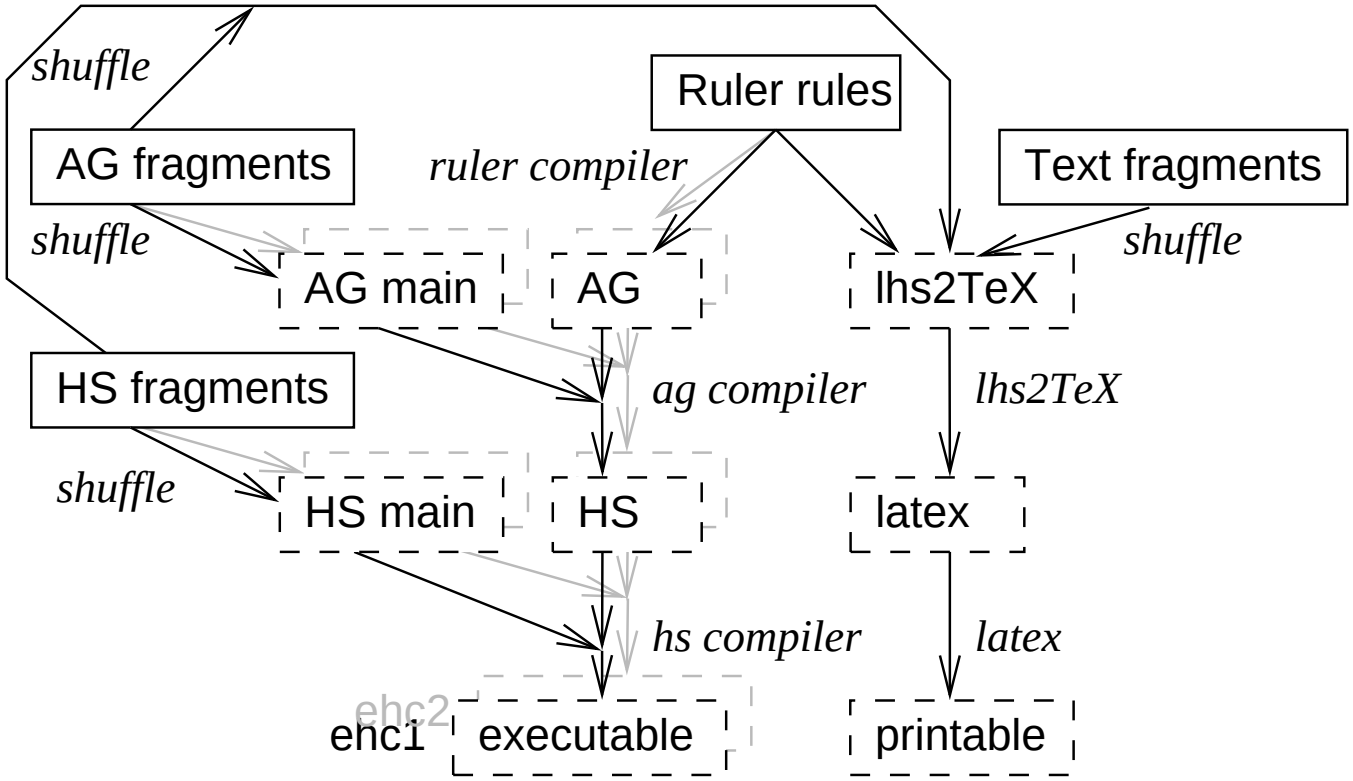
How is it implemented?

- Tools
  - Attribute grammar system
  - Parser combinators
  - Code weaving tools
- Design starting points
  - Stick to standard (Hindley/Milner) type inference
  - Use explicit (type) information provided by programmer
  - And make best effort to propagate this information to where it is needed
  - So we are not limited by type inference

Topics

1. Project/system overview
2. EH versions
3. Code generation
4. Shuffle: fragment logistics
5. Ruler: type rule specification

System overview



: source

: derived

a $\xrightarrow{x}$ b : *b derived from a using x*

grey : *multiple views/compilers*

EH version 1: λ-calculus

- EH program is single expression

$$
\begin{array}{ll}
\textbf{let} & i :: Int \\
 & \quad i = 5 \\
\textbf{in} & i
\end{array}
$$

- Types *Int*, *Char*, tuples and functions

$$
\begin{array}{ll}
\textbf{let} & id :: Int \rightarrow Int \\
 & \quad id = \lambda x \rightarrow x \\
 & \quad fst :: (Int, Char) \rightarrow Int \\
 & \quad fst = \lambda(a, b) \rightarrow a \\
\textbf{in} & id\ (fst\ (id\ 3, \texttt{'x'}))
\end{array}
$$

EH version 1: type checking

- Type signatures are required
- Types are checked

  let i :: *Char*
  $i = 5$
  in i

  gives rise to error annotated representation of program:

```
let i :: Char
    i = 5
    {- ***ERROR(S):
         In '5':
           Type clash:
             failed to fit: Int <= Char
             problem with : Int <= Char -}
    {- [ i:Char ] -}
in i
```

EH version 2: Explicit/implicit typing

- Type signature may be omitted

  $\textbf{let}\ i = 5$
  $\textbf{in}\ \ i$

- Missing type is inferred: $i :: Int$
- Inferred types are monomorphic

  $\textbf{let}\ id = \lambda x \rightarrow x$
  $\textbf{in}\ \ \textbf{let}\ v = id\ 3$
  $\qquad\textbf{in}\ \ id$

  gives rise to type $id :: Int \rightarrow Int$

EH version 3: Polymorphism

- Polymorphism a la Haskell (i.e. Hindley/Milner)
- For example

  $\textbf{let}\ id = \lambda x \rightarrow x$

  $\textbf{in}\ \ id\ 3$

  gives type $id :: \forall\ a.a \rightarrow a$
- Type signature may be given

  $\textbf{let}\ id :: a \rightarrow a$

  $\quad id = \lambda x \rightarrow x$

  $\textbf{in}\ \ id\ 3$
- Type signature can further constrain a type

  $\textbf{let}\ id :: Int \rightarrow Int$

  $\quad id = \lambda x \rightarrow x$

  $\textbf{in}\ \ id\ 3$

EH version 4: Higher ranked types

- Type signatures for quantifiers on argument (higher ranked) positions

  $$\begin{array}{l} \textbf{let}\ f :: (\forall\ a.a \rightarrow a) \rightarrow (Int, Char) \\ \quad\quad f = \lambda i \rightarrow (i\ 3, i\ \text{'x'}) \\ \textbf{in}\ \ f \end{array}$$

- Notational sugaring allows omission of quantifier

  $$\begin{array}{l} \textbf{let}\ f :: (a \rightarrow a) \rightarrow (Int, Char) \\ \quad\quad f = \lambda i \rightarrow (i\ 3, i\ \text{'x'}) \\ \textbf{in}\ \ f \end{array}$$

EH version 4: Existential types

- Existential quantification: hiding/forgetting type information

$$
\begin{array}{ll}
\textbf{let}\ id & :: \forall\ a.a \rightarrow a \\
\quad xy & :: \exists\ a.(a, a \rightarrow Int) \\
\quad xy & = (3, id) \\
\quad ixy & :: (\exists\ a.(a, a \rightarrow Int)) \rightarrow Int \\
\quad ixy & = \lambda(v, f) \rightarrow f\ v \\
\quad xy' & = ixy\ xy \\
\quad pq & :: \exists\ a.(a, a \rightarrow Int) \\
\quad pq & = (\texttt{'x'}, id) \quad \text{-- ERROR} \\
\textbf{in}\ xy'
\end{array}
$$

EH version 4: Existential types

- Notational sugaring allows omission of quantifier
  - $xy :: (a, a \rightarrow Int)$ is interpreted as
  - $xy :: \exists\, a.(a, a \rightarrow Int)$
- Interprets type structure to find suitable location for quantifier
  - a occurs in σ_1 and σ_2 in $\sigma_1 \rightarrow \sigma_2$ and not outside: $\forall$
  - a occurs in σ_1 and σ_2 in (σ_1, σ_2) and not outside: $\exists$

EH version 5: Data types

- User defined data types

$$
\begin{array}{l}
\textbf{let data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a) \\
\textbf{in}\ \ \textbf{let}\ v = \textbf{case}\ \mathit{Cons}\ 3\ \mathit{Nil}\ \textbf{of} \\
\qquad\qquad\qquad \mathit{Nil} \qquad\quad \rightarrow 5 \\
\qquad\qquad\qquad \mathit{Cons}\ x\ y \rightarrow x \\
\qquad\quad \textbf{in}\ \ v
\end{array}
$$

- Unpacking via case expression

EH version 6: Kinds

- Type expressions can be incorrectly used

  let data $List\ a = Nil \mid Cons\ a\ (List\ a)$

  $v :: List$

  in v

- Requires type system for types (similar to type system for values)
- Type of a type: kind
- Examples
  - Kind of $Int :: *$
  - Kind of $List\ a :: *$
  - Kind of $List :: * \rightarrow *$
- Kind inferencing/checking for types (similar to type inferencing/checking for values)
  - Values must have type with kind $::*$

EH version 6: Kind polymorphism

- Kind signatures for types (similar to type signatures for values)

$$
\begin{array}{lll}
\textbf{let}\ Eq & :: k \to k \to * \\
\quad \textbf{data}\ Eq\ a\ b = Eq\ (\forall\ f.f\ a \to f\ b) \\
\quad id & = \lambda x \to x \\
\textbf{in}\ \ \textbf{let}\ g = \textbf{case}\ Eq\ id\ \textbf{of} \\
\qquad\qquad (Eq\ f) \to f \\
\quad \textbf{in}\ \ g
\end{array}
$$

- Polymorphic kinds can also be inferred

$$
\begin{array}{lll}
\textbf{let}\ \textbf{data}\ Eq\ a\ b = Eq\ (\forall\ f.f\ a \to f\ b) \\
\quad id & = \lambda x \to x \\
\textbf{in}\ \ \textbf{let}\ v = \textbf{case}\ Eq\ id\ \textbf{of} \\
\qquad\qquad Eq\ f \to f \\
\quad \textbf{in}\ \ v
\end{array}
$$

infers kind `Eq :: Forall a . a -> a -> *`

EH version 7: Non extensible records

- Replacement for tuples

$$
\begin{array}{l}
\textbf{let}\ r = (i = 3, c = \texttt{'x'}, id = \lambda x \rightarrow x) \\
\quad\quad s = (r \mid c := 5) \\
\textbf{in}\ \ \textbf{let}\ v = (r.id\ r.i, r.id\ r.c) \\
\quad\quad\quad\quad vi = v.1 \\
\quad\quad \textbf{in}\ \ vi
\end{array}
$$

EH version 8: Code generation

- In phases
  - to core representation (removing syntactic sugar, ...)
  - via transformations (lambda lifting, ...)
  - to code for abstract sequential machine
- Interpreter for abstract sequential machine

EH version 9: Class system, explicit implicit parameters

- Class system
- + named instances
- + explicit dictionary passing
- + scoping for instances
- + coercions

EH version 9

- Being explicit using implicit mechanism

$$
\begin{aligned}
&\mathit{nub} && :: (\mathit{Eq}\ a) \Rightarrow [a] \rightarrow [a] \\
&\mathit{nub}\ [\,] && = [\,] \\
&\mathit{nub}\ (h : t) && = h : \mathit{nub}\ (\mathit{filter}\ (\mathit{not}.(h ==))\ t) \\
&\mathit{nubBy} && :: (a \rightarrow a \rightarrow \mathit{Bool}) \rightarrow [a] \rightarrow [a] \\
&\mathit{nubBy}\ \mathit{eq} && = \mathit{nub}\{\,!((==) = \mathit{eq}) <: \mathit{Eq}\ \mathit{Int}!\,\} \\
&t_1 = \mathit{nub}\ [3, 3, 4, 5] \\
&t_2 = \mathit{nub}\{\,!((==) = \ldots) <: \mathit{Eq}\ \mathit{Int}!\,\}[3, 3, 4, 5]
\end{aligned}
$$

EH version 10: Extensible records

- Flexibility w.r.t. presence of labels

$$
\begin{array}{l}
\textbf{let}\ \mathit{add} :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \\
\quad f \quad :: (r\lambda x, r\lambda y) \Rightarrow (r \mid x :: \mathit{Int}, y :: \mathit{Int}) \rightarrow \mathit{Int} \\
\quad f \quad = \lambda r \rightarrow \mathit{add}\ r.x\ r.y \\
\textbf{in} \\
\textbf{let}\ v_1 = f\ (x = 3, y = 4) \\
\quad v_2 = f\ (y = 5, a = \texttt{'z'}, x = 6) \\
\textbf{in}\ \ v_2
\end{array}
$$

- More general tuple access

$$
\begin{array}{l}
\textbf{let}\ \mathit{snd} = \lambda r \rightarrow r.2 \\
\textbf{in} \\
\textbf{let}\ v_1 = \mathit{snd}\ (3, 4) \\
\quad v_2 = \mathit{snd}\ (3, 4, 5) \\
\textbf{in}\ \ v_2
\end{array}
$$

1 Project/system overview

2 EH versions

3 Code generation

4 Shuffle: fragment logistics

5 Ruler: type rule specification

Code generation example to GRIN

- Compilation to imperative (non-lazy) machine
  - to Graph Reduction Intermediate Notation (GRIN)
  - all lazy constructs are coded imperatively
  - intermediate imperative notation translated to `C` (`C--`)
- Example

$$
\begin{array}{l}
\textbf{let data } \mathit{Tup} = \mathit{Tup}\ \mathit{Int}\ \mathit{Int} \\
\quad \mathit{snd} = \lambda(\mathit{Tup}\ x\ y) \rightarrow y \\
\quad \mathit{app} = \lambda f\ x \rightarrow f\ x \\
\textbf{in}\ \ \mathit{snd}\ (\mathit{app}\ (\mathit{Tup}\ 1)\ 42)
\end{array}
$$

Intermediate core language

- Explicit strictness, ...

```
let  Tup = \x1 -> \x2 -> #(0/Tup/2) (x1) (x2)
in
 let  app = \f -> \x -> f (x)
 in
  let  snd = \t ->  let ! t!_0 = t
                    in  case t!_0 of
                          (0/Tup/2 x y))
                             -> y
                          _  -> undefined
  in
   let  5_0_22 = Tup (1)
   in
    let  5_0_20 = app (5_0_22) (42)
    in
     let  main = snd (5_0_20)
     in
      main
```

- Explicit evaluation and application

```
  _Tup x1 x2
    = { unit (#0/C/_Tup x1 x2)}
; _app f x
    = { eval f ; \f' ->
        apply f' x
      }
; _snd t
    = { unit t ; \t' ->
        eval t' ; \t'' ->
        case t'' of
          { (#0/C/_Tup x y)
                -> { unit y ; \y' ->
                     eval y'
                   }
          }
      }
; $_5_0_22
    = { store (#0/C/$_Int 1 1) ; \i ->
        unit (#0/P/1/_Tup 1 i)
      }
; $_5_0_20
    = { store (#0/C/$_Int 1 42) ; \i ->
        _app $_5_0_22 i
      }
; main
    = { _snd $_5_0_20}
```

- Explicit evaluation and application

```
; apply f a
    = { case f of
          { (#0/P/1/_Tup x)
                -> { _Tup x a}
          ; (#0/P/2/_Tup)
                -> { unit (#0/P/1/_Tup a)}
          }
      }
; eval v
    = { fetch v   ; \v' ->
        case v' of
          { (#0/C/_Tup _ _)
                -> { unit v'}
          ; (#0/P/1/_Tup _)
                -> { unit v'}
          ; (#0/P/2/_Tup _)
                -> { unit v'}
          ; (#0/F/_Tup x y)
                -> { _Tup x y ; \t ->
                     update v t  ; \() ->
                     unit t
                   }
          ; (#0/F/_app x y)
                -> { _app x y ; \a ->
                     update v a  ; \() ->
                     unit a
                   }
          }
      }
```

1 Project/system overview

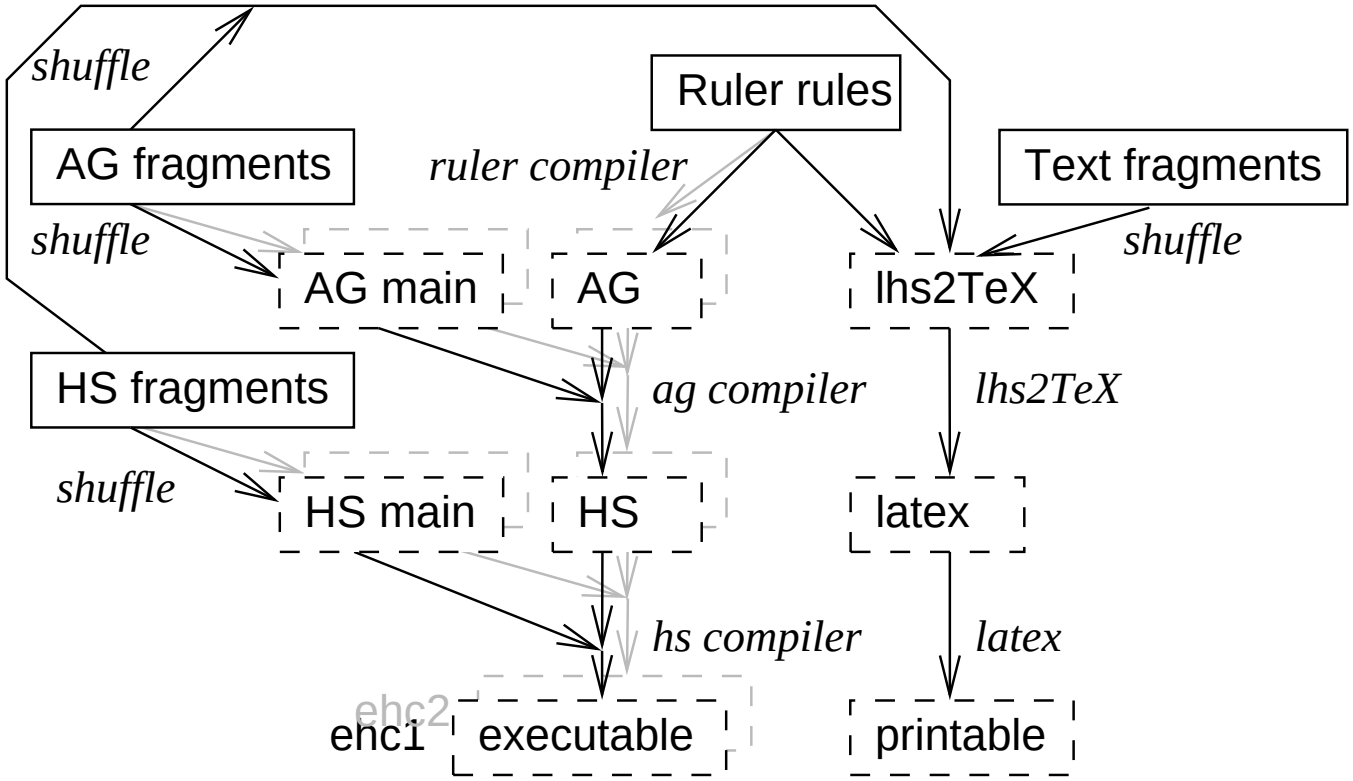
2 EH versions

3 Code generation

4 Shuffle: fragment logistics

5 Ruler: type rule specification

System overview



: *source*

: *derived*

a $\xrightarrow{x}$ b : *b derived from a using x*

grey : *multiple views/compilers*

Shuffle: the problem

- Given some program fragments
  - AG:

```
SEM Expr
  | Int  lhs  .  ty  =  Ty_Int
```

- How to ensure consistent inclusion in its explanation:
  - "the following code ...:

    sem *Expr*

    | *Int* **lhs**.*ty* = *Ty_Int*

    ... this code ..."
- The problem: no copy-and-paste allowed
- The solution: given names to fragments
  - similar to literate programming
  - but mixing of code and explanation does not help programming
- For all languages: Haskell, ...

The problem

- Given subsequent versions
  - construction of type variable, version 2 (Haskell):

    $mkTyVar :: TyVarId \rightarrow Ty$
    $mkTyVar\ tv = Ty_Var\ tv$
  - version 3:

    $mkTyVar :: TyVarId \rightarrow Ty$
    $mkTyVar\ tv = Ty_Var\ tv\ TyVarCateg_Plain$
- How to avoid duplicate definitions?
- The problem: different versions require different definitions
- The solution: overriding mechanism, notion of views + inheritance
  - similar to OO programming
  - but for code fragments

Example

- Two versions, in file `EHTy.chs`

```
%%[2.mkTyVar
mkTyVar :: TyVarId -> Ty
mkTyVar tv = Ty_Var tv
%%]

%%[3.mkTyVar -2.mkTyVar
mkTyVar :: TyVarId -> Ty
mkTyVar tv = Ty_Var tv TyVarCateg_Plain
%%]
```

- Referred to
  - via *Shuffle* by `%%@EHTy.2.mkTyVar`
  - via LaTeX by `\chunkCmdRef{EHTy.2.mkTyVar}`

1 Project/system overview

2 EH versions

3 Code generation

4 Shuffle: fragment logistics

5 Ruler: type rule specification

Motivation

- Experimenting with programming language design and implementation where many features are combined
  - Essential Haskell (EHC project)

Motivation

- Experimenting with programming language design and implementation where many features are combined
  - Essential Haskell (EHC project)
- Stepwise organisation
  - EH1: λ-calculus & type checking
  - EH2: Explicit/implicit typing
  - EH3: Polymorphism
  - EH4: Higher ranked types & existential types
    - & experimental quantifier propagation
  - EH5: Data types
  - EH6: Kinds
  - EH7: Non extensible records
  - EH8: Code generation (to GRIN)
  - EH9: Class system & explicit implicit parameters
  - EH10: Extensible records

Motivation

- Example experiment: exploitation of type signatures

$$
\begin{array}{l}
\textbf{let}\ g\ ::\ (\forall\ a.a \rightarrow a)\ \rightarrow Int \\
\quad id = \lambda x \rightarrow x \\
\quad f\ \ = \lambda h \rightarrow \textbf{let}\ x_1 = h\ 3 \\
\qquad\qquad\qquad\quad\ \ x_2 = h\ \text{'x'} \\
\qquad\qquad\qquad\quad\ \ y\ \ = g\ h \\
\qquad\qquad\qquad\ \ \textbf{in}\ \ x_1 \\
\textbf{in}\ \ f\ id
\end{array}
$$

Motivation

- Example experiment: exploitation of type signatures

$$
\begin{array}{l}
\textbf{let}\ g\ ::\ (\forall\ a.a \rightarrow a)\ \rightarrow Int \\
\quad id = \lambda x \rightarrow x \\
\quad f\ \ = \lambda h \rightarrow \textbf{let}\ x_1 = h\ 3 \\
\qquad\qquad\qquad\qquad\ \ x_2 = h\ \text{'x'} \\
\qquad\qquad\qquad\qquad\ \ y\ \ = g\ h \\
\qquad\qquad\qquad\ \ \textbf{in}\ \ x_1 \\
\textbf{in}\ \ f\ id
\end{array}
$$

- Our desire: combine a large number of such features
  - *and* keep the implementation maintainable, understandable

Our goal

- To deal with such features in isolation in a cleanroom situation
  - but how do they interact in combination?

Our goal

- To deal with such features in isolation in a cleanroom situation
  - but how do they interact in combination?
- To combine such features
  - but combination of features easily leads to entanglement in description and implementation,
  - so how do we achieve separation across specification and implementation?

Our goal

- To deal with such features in isolation in a cleanroom situation
  - but how do they interact in combination?
- To combine such features
  - but combination of features easily leads to entanglement in description and implementation,
  - so how do we achieve separation across specification and implementation?
- To separately describe specification, explanation and implementation
  - but how do we achieve mutual consistency?

Motivational example

- Suppose you have a language

$e ::=$	int	literals
$\mid$	i	program variable
$\mid$	$e\ e$	application
$\mid$	$\lambda i \rightarrow e$	abstraction
$\mid$	$\textbf{let}\ i = e\ \textbf{in}\ e$	local definitions

- With a type system specification (partially given)

$$\frac{i \mapsto \sigma \in \Gamma \qquad \tau = inst\ (\sigma)}{\Gamma \vdash^{e} i : \tau}\ \mathrm{E.VAR}_E \qquad \frac{\Gamma \vdash^{e} a : \tau_a \qquad \Gamma \vdash^{e} f : \tau_a \rightarrow \tau}{\Gamma \vdash^{e} f\ a : \tau}\ \mathrm{E.APP}_E$$

Two formalisms

- Rule & Attribute Grammar (AG) implementation

$$\frac{i \mapsto \sigma \in \Gamma \qquad \tau = \mathit{inst}\ (\sigma)}{\Gamma \vdash^{e} i : \tau}\ \mathrm{E.VAR}_{E}$$

```
data Expr
    | Var i : {String}
attr Expr [g : Gam | c : C | ty : Ty]
sem Expr
    | Var (lhs.uniq, loc.uniq1)
                = rulerMk1Uniq @lhs.uniq
          (loc.pty_, loc.nmErrs)
                = gamLookup @i @lhs.g
          lhs.ty = tyInst @uniq1 @pty_
```

- Mutual consistency is relatively easy to see

Motivational example

The problem is complexity, arising as a result of feature combination

- EHC: impredicativity propagation algorithm (for application)

$$\boxed{o; \Gamma; \mathbb{C}^k; \mathcal{C}^k; \sigma^k \vdash^e e : \mathfrak{o}; \sigma \rightsquigarrow \mathbb{C}; \mathcal{C}}$$

$$\frac{\begin{array}{c} v \text{ fresh} \\ o_{str}; \Gamma; \mathbb{C}^k; \mathcal{C}^k; v \to \sigma^k \vdash^e e_1 : \mathfrak{o}_f; _ \to \sigma \rightsquigarrow \mathbb{C}_f; \mathcal{C}_f \\ o_{im} \vdash^{\leqslant} \mathfrak{o}_f \leqslant \mathbb{C}_f(v \to \sigma^k) : _ \rightsquigarrow \mathbb{C}_F \\ o_{inst-lr}; \Gamma; \mathbb{C}_F\mathbb{C}_f; \mathcal{C}_f; v \vdash^e e_2 : \mathfrak{o}_a; _ \rightsquigarrow \mathbb{C}_a; \mathcal{C}_a \\ fi^{+}_{alt}, o_{inst-l} \vdash^{\leqslant} \mathfrak{o}_a \leqslant \mathbb{C}_a v : _ \rightsquigarrow \mathbb{C}_A \\ \mathbb{C}_1 \equiv \mathbb{C}_A\mathbb{C}_a \end{array}}{o; \Gamma; \mathbb{C}^k; \mathcal{C}^k; \sigma^k \vdash^e e_1\ e_2 : \mathbb{C}_1\sigma^k; \mathcal{C}_a\sigma \rightsquigarrow \mathbb{C}_1; \mathcal{C}_a} \ \text{E.APP}_{l2}$$

(Just for the idea)

```
sem Expr
  | App (func.gUniq, loc.uniq1, loc.uniq2, loc.uniq3)
                                  = mkNewLevUID3 @lhs.gUniq
        loc  .tvarv_              = mkTyVar @uniq1
        func.fiOpts               = o_str
             .knTy                = [@tvarv_] `mkArrow` @lhs.knTy
        (_, loc.ty_)              = tyArrowArgRes @func.ty
        loc  .fo_fitF_            = fitsIn o_im @fe @uniq2 @func.imprTy (@func.imprTy
        arg  .fiOpts              = o_inst-lr
             .knTy                = @tvarv_
             .imprTyCnstr         = foCnstr @fo_fitF_ ⊕ @func.imprTyCnstr
        loc  .fo_fitA_            = fitsIn (o_inst-l{fi_alt = True}) @fe @uniq3 @arg.imprT
             .imprTyCnstr_1_      = foCnstr @fo_fitA_ ⊕ @arg.imprTyCnstr
             .imprTy              = @imprTyCnstr_1_ ⊕ @lhs.knTy
             .ty                  = @arg.tyCnstr ⊕ @ty_
        lhs  .imprTyCnstr         = @imprTyCnstr_1_
```

Views

- Specify subsequent views, building on top of eachother, starting with the first:

$$\frac{\begin{array}{c} i \mapsto \sigma \in \Gamma \\ \tau = \mathit{inst}\ (\sigma) \end{array}}{\Gamma \vdash^{e} i : \tau}\ \mathrm{E.VAR}_{E}$$

- And then the differences relative to the previous

$$\frac{\begin{array}{c} i \mapsto \sigma \in \Gamma \\ \tau = \mathit{inst}\ (\sigma) \end{array}}{\mathcal{C}^{k}; \Gamma \vdash^{e} i : \tau \rightsquigarrow \mathcal{C}^{k}}\ \mathrm{E.VAR}_{A}$$

Specification

- Structure/scheme for judgements

  scheme *expr* =
  view E =
  holes $[e : Expr, gam : Gam, ty : Ty]$
  judgespec $gam \vdash e : ty$
  judgeuse tex $gam \vdash$ ..`"e"` $e : ty$
  view A =
  holes [**inh** $gam : Gam$, **thread** $cnstr : \mathcal{C}$, **syn** $ty : Ty$]
  judgespec $cnstr$.**inh**; $gam \vdash e : ty \rightsquigarrow cnstr$.**syn**
  judgeuse tex $cnstr$.**inh**; $gam \vdash$ ..`"e"` $e : ty \rightsquigarrow cnstr$.**syn**

Judgements in rules

- Instantiation of scheme

```
ruleset expr.base scheme expr "Expression type rules" =
  rule e.var =
    view E =
      judge G : gamLookupIdTy = i ↦ pty ∈ gam
      judge I : tyInst = ty `=` inst (pty)
       –
      judge R : expr = gam ⊢ i : ty
    view A =
       –
      judge R : expr
          | cnstr.syn = cnstr.inh
```

AG code generation

- Translation to AG

```
sem Expr
  | Var (lhs.uniq, loc.uniq1)
                = rulerMk1Uniq @lhs.uniq
        (loc.pty_, loc.nmErrs)
                = gamLookup @i @lhs.g
        lhs.ty = tyInst @uniq1 @pty_
```

- (With additional specification for freshness and lookup)

And then ...

- Develop into full Haskell compiler
- Platform for experimentation/education
  - Haskell itself: language design, implementation, optimisation, ...
  - Meta: tools (like *Ruler*, AG, ...) for specifying the implementation
  - Integration with other techniques (e.g. Helium's error reporting)
- More info
  - `http://www.cs.uu.nl/groups/ST/Ehc/WebHome`