

Essential Haskell Compiler overview

Atze Dijkstra and S. Doaitse Swierstra

January 7, 2005

Why and what?

Purpose

A short tour

- ▶ 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

And then?

- ▶ Extend the set of compilers up to a full Haskell++ compiler
- ▶ Keep it understandable
- ▶ So it can still be used for experimentation and education
- ▶ Will grow over time
 - ▶ see <http://www.cs.uu.nl/groups/ST/Ehc/WebHome>

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

EH version 1: λ -calculus

- ▶ EH program is single expression

```
let i :: Int
    i = 5
in i
```
- ▶ Types *Int*, *Char*, tuples and functions

```
let id :: Int → Int
    id = λx → x
    fst :: (Int, Char) → Int
    fst = λ(a, b) → a
in id (fst (id 3, 'x'))
```

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
let $i = 5$
in i
- ▶ Missing type is inferred: $i :: Int$
- ▶ Inferred types are monomorphic
let $id = \lambda x \rightarrow x$
in let $v = id\ 3$
in id
gives rise to type $id :: Int \rightarrow Int$

EH version 3: Polymorphism

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

- ▶ For example

```
let id =  $\lambda x \rightarrow x$ 
```

```
in id 3
```

gives type $id :: \forall a. a \rightarrow a$

- ▶ Type signature may be given

```
let id ::  $a \rightarrow a$ 
```

```
    id =  $\lambda x \rightarrow x$ 
```

```
in id 3
```

- ▶ Type signature can further constrain a type

```
let id ::  $Int \rightarrow Int$ 
```

```
    id =  $\lambda x \rightarrow x$ 
```

```
in id 3
```

EH version 4: Higher ranked types

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

```
let f :: (∀ a.a → a) → (Int, Char)
```

```
    f = λi → (i 3, i 'x')
```

```
in f
```

- ▶ Notational sugaring allows omission of quantifier

```
let f :: (a → a) → (Int, Char)
```

```
    f = λi → (i 3, i 'x')
```

```
in f
```

EH version 4: Existential types

- ▶ Existential quantification: hiding/forgetting type information

```
let id  ::  $\forall a. a \rightarrow a$   
    xy  ::  $\exists a. (a, a \rightarrow Int)$   
    xy  = (3, id)  
    ixy ::  $(\exists a. (a, a \rightarrow Int)) \rightarrow Int$   
    ixy =  $\lambda(v, f) \rightarrow f\ v$   
    xy' = ixy xy  
    pq  ::  $\exists a. (a, a \rightarrow Int)$   
    pq  = ('x', id)  -- ERROR  
in xy'
```

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

```
let data List a = Nil | Cons a (List a)
```

```
in let v = case Cons 3 Nil of
```

```
    Nil      → 5
```

```
    Cons x y → x
```

```
in v
```

- ▶ Unpacking via case expression

- ▶ Type expressions can be incorrectly used
let data *List a = Nil | 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*:: * → *
- ▶ Kind inferencing/checking for types (similar to type inferencing/checking for values)
 - ▶ Values must have type with kind ::*

EH version 6: Kind polymorphism

- ▶ Polymorphic kinds

```
let data Eq a b = Eq (∀ f.f a → f b)
```

```
  id           = λx → x
```

```
in let v = case Eq id of
```

```
      Eq f → f
```

```
  in v
```

```
inferred kind Eq :: forall a . a -> a -> *
```

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

```
let Eq           :: k → k → *
```

```
  data Eq a b = Eq (∀ f.f a → f b)
```

```
  id           = λx → x
```

```
in let g = case Eq id of
```

```
      (Eq f) → f
```

```
  in g
```

EH version 7: Non extensible records

► Replacement for tuples

```
let  $r = (i = 3, c = 'x', id = \lambda x \rightarrow x)$ 
```

```
     $s = (r \mid c:=5)$ 
```

```
in let  $v = (r.id\ r.i, r.id\ r.c)$ 
```

```
     $vi = v.1$ 
```

```
in  $vi$ 
```

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

```

let data List a = Nil | Cons a (List a)  -- prelude
data Bool = False | True
  ¬ :: Bool → Bool
  filter :: (a → Bool) → List a → List a
class Eq a where
  eq :: a → a → Bool
  ne :: a → a → Bool
instance dEqInt <~ Eq Int where
  eq = primEqInt
  ne = λx y → ¬ (eq x y)
in let nub :: Eq a ⇒ List a → List a
  nub = λxx → case xx of
    Nil          → Nil
    Cons x xs    → Cons x (nub (filter (ne x) xs))
in nub (#(dEqInt | eq:=λx y → ...) <~ Eq Int#)
  (Cons 3 (Cons 3 (Cons 4 Nil)))

```

EH version 10: Extensible records

- ▶ Flexibility w.r.t. presence of labels

```
let add :: Int → Int → Int
```

```
    f    :: (rλx, rλy) ⇒ (r | x :: Int, y :: Int) → Int
```

```
    f    = λr → add r.x r.y
```

```
in
```

```
let v1 = f (x = 3, y = 4)
```

```
    v2 = f (y = 5, a = 'z', x = 6)
```

```
in v2
```

- ▶ More general tuple access

```
let snd = λr → r.2
```

```
in
```

```
let v1 = snd (3, 4)
```

```
    v2 = snd (3, 4, 5)
```

```
in v2
```

EH version [11..]: ...

- ▶ (Student) projects
 - ▶ Support for Attribute Grammars
 - ▶ Efficient code generation