Advanced Functional Programming 2011-2012, period 2

Andres Löh and Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

Jan 12, 2012

13. More Types, Lambda Cube



This lecture

More Types, Lambda Cube

The Equality type

Higher-rank polymorphism

The rank of a type

Impredicativity

Systems F, F_{ω} , FC

The lambda cube

 F_{ω}

From F_{ω} to GHC's core



13.1 The Equality type



Comparing the length of vectors

equalLength :: Vec a m ightarrow Vec b n ightarrow Bool



Comparing the length of vectors

equalLength :: Vec a m ightarrow Vec b n ightarrow Bool

Not useful, because

```
 \begin{tabular}{ll} \textbf{if} \ equalLength} \ v \ w \ \textbf{then} \ head} \ (zipWith \ (,) \ v \ w) \\ \textbf{else} \ \ \dots \\ \end{tabular}
```

will not type check. We loose the information that m and n are equal because we return only a Bool!

4日 > 4 個 > 4 豆 > 4 豆 > 豆 めの()

Equality type

data Equal :: $* \rightarrow * \rightarrow *$ where Refl :: Equal a a



◆ロト→園ト→夏ト→夏ト 夏 夕へ○

Equality type

```
data Equal :: * \rightarrow * \rightarrow * where
  Refl:: Equal a a
equalLength :: Vec a m \rightarrow Vec b n \rightarrow Maybe (Equal m n)
equalLength Nil
                                 Nil = Just Refl
equalLength (Cons x xs) (Cons y ys) =
  case equalLength xs ys of
     Just Refl
                                          \rightarrow Just Refl
     Nothing
                                          \rightarrow Nothing
equalLength _
                                           = Nothing
```

4□▶
4□▶
4□▶
4□▶
4□
5
9
0

Equality type

```
data Equal :: * \rightarrow * \rightarrow * where
  Refl:: Equal a a
equalLength :: Vec a m \rightarrow Vec b n \rightarrow Maybe (Equal m n)
equalLength Nil
                                  Nil = Just Refl
equalLength (Cons x xs) (Cons y ys) =
  case equalLength xs ys of
     Just Refl
                                           \rightarrow Just Refl
     Nothing
                                           \rightarrow Nothing
equalLength _
                                           = Nothing
test :: Vec a m \rightarrow Vec b (Succ n) \rightarrow (a, b)
test v w = case equalLength v w of
                Just Refl \rightarrow head (zipWith (, ) v w)
```

Expressive power of equality

The equality type can be used to encode nearly all other GADTs:

```
data Expr :: * \rightarrow * where
                                                   \rightarrow Expr Int
   If :: \mathsf{Expr}\:\mathsf{Bool} \to \mathsf{Expr}\:\mathsf{a} \to \mathsf{Expr}\:\mathsf{a} \to \mathsf{Expr}\:\mathsf{a}
  Pair :: Expr a \rightarrow Expr b \rightarrow Expr (a,b)

ightarrow Expr a
   Fst :: Expr (a, b)
data Expr t =
            Int (Equal t Int) Int
                  (Expr Bool) (Expr t) (Expr t)
    | ∀a b.Pair (Equal t (a, b)) (Expr a) (Expr b)
    | \forall a \text{ b.Fst (Equal t a)}  (Expr (a, b))
```

◆□▶◆御▶◆団▶◆団▶ 団 めの◎

Outlook Generic programming: Reflecting types

```
data Type :: * \rightarrow * where
Int :: Type Int
Bool :: Type Bool
List :: Type a \rightarrow Type [a]
Pair :: Type a \rightarrow Type b \rightarrow Type (a, b)
```

We can now write generic functions as functions of the form

```
\mathsf{f} :: \mathsf{Type} \ \mathsf{a} 	o \ldots \mathsf{a} \ldots
```

and define dynamic values by packing up a type representation with a value

```
data Dynamic :: * where

Dyn :: Type a \rightarrow a \rightarrow Dynamic
```



Summary

- ► GADTs can be used to encode advanced properties of types in the type language.
- We end up mirroring expression-level concepts on the type level (e.g. natural numbers).
- GADTs can also represent data that is computationally irrelevant and just guides the type checker (equality proofs, evidence for addition).
 - Such information could ideally be erased, but in Haskell, we can always cheat via \perp :
 - ot :: Equal Int Bool



13.2 Higher-rank polymorphism





State

The ST monad is a restricted form of IO. It offers mutable references, but no other IO features. It can therefore be "run": unlike IO, we can escape from ST without having to use anything unsafe.

```
data ST s a -- abstract data STRef s a -- abstract newSTRef :: a \rightarrow ST s (STRef s a) readSTRef :: STRef s a \rightarrow ST s a writeSTRef :: STRef s a \rightarrow a \rightarrow ST s () runST :: (\forall s.ST s a) \rightarrow a
```

We can only run an ST monad if it is polymorphic in s. Why?



◆□▶◆御▶◆団▶◆団▶ 団 めの◎

Preventing escaping references

This should fail:

```
\begin{array}{l} t_1 :: \mathsf{ST} \ \mathsf{s}_1 \ (\mathsf{STRef} \ \mathsf{s}_1 \ \mathsf{Int}) \\ t_1 = \mathsf{newSTRef} \ 0 \\ t_2 :: \mathsf{STRef} \ \mathsf{s}_2 \ \mathsf{Int} \to \mathsf{ST} \ \mathsf{s}_2 \ \mathsf{Int} \\ t_2 \ \mathsf{ref}_2 = \mathsf{readSTRef} \ \mathsf{ref}_2 \\ t_3 :: \mathsf{Int} \\ t_3 = \mathsf{runST} \ (\mathsf{t}_2 \ (\mathsf{runST} \ \mathsf{t}_1)) \end{array}
```

- ▶ One state thread introduces a mutable variable.
- ▶ The "address" is passed to another.
- ► The variable is read there.



Why should it fail?

- Since runST allows to escape from the ST monad, several runST calls are not sequenced and can be run in parallel.
- ▶ If different ST computations can use each other's mutable variables, it is unclear what will happen when, thereby breaking referential transparency.

Why does it fail?

- ► Think of a computation of type ST s a as a state thread with a tag of type s and a result of type a.
- ► When a state thread is run, it is assigned a specific, unique tag.
- ▶ Different state threads (i.e., different calls to runST) get different tags.
- ► But we (the programmers) do not know which tag, so we are trying to write ST computations such that they are **polymorphic** in the tag.
- ► The function runST then requires the passed ST computation to be polymorphic in the tag:

 $\mathsf{runST} :: (\forall \mathsf{s.ST} \; \mathsf{s} \; \mathsf{a}) \to \mathsf{a}$



Why does it fail (contd.)

```
\begin{aligned} &t_1 :: \mathsf{ST} \; \mathsf{s}_1 \; (\mathsf{STRef} \; \mathsf{s}_1 \; \mathsf{Int}) \\ &t_1 = \mathsf{newSTRef} \; 0 \\ &t_2 :: \mathsf{STRef} \; \mathsf{s}_2 \; \mathsf{Int} \to \mathsf{ST} \; \mathsf{s}_2 \; \mathsf{Int} \\ &t_2 \; \mathsf{ref}_2 = \mathsf{readSTRef} \; \mathsf{ref}_2 \\ &t_3 :: \mathsf{Int} \\ &t_3 = \mathsf{runST} \; (\mathsf{t}_2 \; (\mathsf{runST} \; \mathsf{t}_1)) \end{aligned}
```

- ▶ The function t₁ is polymorphic in s₁.
- ▶ When run, t₁ is instantiated at a particular type, let's call it tag.
- ► The result of t₁ is thus STRef tag Int, which can only be read resulting in an ST tag Int value



13.3 The rank of a type



Rank-1 polymorphism

► Typical Haskell types are quantified implicitly at the outer level. The type signature

$$\mathsf{map} :: (\mathsf{a} \to \mathsf{b}) \to [\mathsf{a}] \to [\mathsf{b}]$$

is an abbreviation for

$$\mathsf{map} :: \forall \mathsf{a} \; \mathsf{b}. (\mathsf{a} \to \mathsf{b}) \to [\mathsf{a}] \to [\mathsf{b}]$$

Rank-n polymorphism

▶ If a function is parameterized over a polymorphic function, then the rank of its type goes up by 1:

$$\mathsf{runST} :: \forall \mathsf{a}. (\forall \mathsf{s}.\mathsf{ST} \mathsf{\ s\ a}) \to \mathsf{a}$$

Here, the argument is rank-1 polymorphic, so runST is said to be rank-2 polymorphic.

- ► There's no limit to the rank. A function can for example be parameterized by a rank-2 polymorphic function and then is itself rank-3 polymorphic.
- ▶ Requires RankNTypes. The use of the inner ∀ is not optional.

The difference

Compare:

 $\begin{array}{c} \mathsf{runST} \,:: \forall \mathsf{a}. (\forall \mathsf{s.ST} \, \mathsf{s} \, \mathsf{a}) \to \mathsf{a} \\ \mathsf{runST'} :: \forall \mathsf{a} \, \mathsf{s.} \quad \mathsf{ST} \, \mathsf{s} \, \mathsf{a} \ \to \mathsf{a} \end{array}$

4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□
9
0

The difference

Compare:

```
runST :: \foralla.(\foralls.ST s a) \rightarrow a runST' :: \foralla s. ST s a \rightarrow a
```

- ► For runST′, the caller may choose both a and s arbitrarily.
- ► For runST, the caller may choose a, but the callee (the implementor of runST) may choose s.

Other uses

- Polymorphic components (as discussed in the context of evidence translation for type classes).
- ► Functions on nested datatypes (as seen in the weekly assignments).

◆□▶◆御▶◆団▶◆団▶ 団 めの◎

Higher rank and type inference

As with many other extensions, the use of higher-rank polymorphism requires explicit type annotations.

- ► A function with higher-rank polymorphic type must have an explicit type signature.
- ► The application of a function with higher-rank polymorphic type typically does not require a type annotation.



13.4 Impredicativity



What do quantifiers quantify over?

► Higher-rank polymorphism allows quantifiers (and class constraints) deep in the types, but what do quantified type variables range over?

What do quantifiers quantify over?

- ► Higher-rank polymorphism allows quantifiers (and class constraints) deep in the types, but what do quantified type variables range over?
- Normal answer: type variables can only be instantiated to monomorphic types of the correct kind. With this restriction, a type system is called predicative.

What do quantifiers quantify over?

- ► Higher-rank polymorphism allows quantifiers (and class constraints) deep in the types, but what do quantified type variables range over?
- Normal answer: type variables can only be instantiated to monomorphic types of the correct kind. With this restriction, a type system is called predicative.
- Impredicative polymorphism lifts this restriction (ImpredicativeTypes).

Data structures containing polymorphic values

► The most interesting application of impredicativity is instantiating datatype parameters to polymorphic types. Example:

$$[\mathsf{runST}] :: [(\forall \mathsf{s.ST} \; \mathsf{s} \; \mathsf{a}) \to \mathsf{a}]$$

► Unfortunately, the current implementation of impredicativity in GHC makes it hardly usable in practice, because it requires lots of type annotations.

14. Systems F, F_{ω} , FC



Recap

- Most language extensions can be translated into a core language.
- ▶ Most compilers make use of a core language internally, because that simplifies the implementation of optimizations and code generation.

GHC's core

GHC makes use of an explicitly typed core language.

- ► Advantage: Useful sanity check for type system extensions and optimizations.
- ▶ Disadvantage: Some extensions could easily be translated into an untyped lambda calculus, but don't necessarily fit into the selected core language.

GHC's core (contd.)

- ▶ GHC's core is a variant of the calculus known as F_{ω} . It supports
 - Arbitrary rank polymorphism.
 - ► Types of arbitrary kinds.
 - Impredicativity.
- Many features have been added to the core, such as
 - Datatypes and (simple) pattern matching.
 - Foreign function interface.
 - Type equality constraints (recent, for GADTs and type families; called System FC).

14.1 The lambda cube

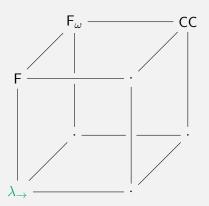




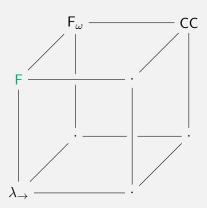
The lambda cube

- ightharpoonup GHC's base system F_{ω} is a well-studied system.
- ▶ Part of Barendregt's lambda cube.
- ► The lambda cube classifies different typed lambda calculi on three dimensions of possible lambda abstraction:
 - terms depending on types (polymorphism)
 - types depending on types (higher-kinded types)
 - types depending on terms (dependent types)

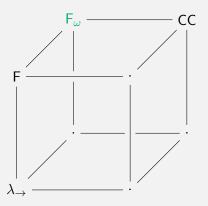




 $\lambda_{
ightarrow}$ – the simply typed lambda calculus has no polymorphism and no kind system

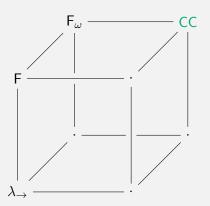


F – **System F** has polymorphism (also higher-rank) but no kinds



 F_{ω} – **System F** $_{\omega}$ has polymorphism and kinds

Faculty of Science



CC – the **calculus of constructions** forms the basis of dependently typed programming languages such as Coq or Agda

14.2 F_{ω}



Syntax

Expressions:

$$\begin{array}{|c|c|c|c|}\hline e ::= x & -- \ variable \\ & \mid \ \lambda(x :: t) \rightarrow e & -- \ abstraction \\ & \mid \ (e \ e) & -- \ application \\ & \mid \ \Lambda(a :: K) \rightarrow e & -- \ type \ abstraction \\ & \mid \ (e \ \langle t \rangle) & -- \ type \ application \\ \end{array}$$

Note how all lambda abstractions are explicitly typed.

Syntax (contd.)

Types:

Kinds:

$$\mathsf{K} ::= *$$
 -- base kind $\mathsf{K} \to \mathsf{K}$ -- function kind

Note that there is no polymorphism on the kind-level, like in Haskell.



Syntax examples

Polymorphic identity:

$$\Lambda(\mathsf{a}::*) o \lambda(\mathsf{x}::\mathsf{a}) o \mathsf{x}$$

Applying the identity to a Bool (assuming we have Bool):

$$(\Lambda(\mathsf{a} :: *) \to \lambda(\mathsf{x} :: \mathsf{a}) \to \mathsf{x}) \langle \mathsf{Bool} \rangle \mathsf{ False}$$

Map on lists (assuming we have lists and case on lists):

$$\left| \begin{array}{l} \Lambda(\mathsf{a} :: *) \; (\mathsf{b} :: *) \to \lambda(\mathsf{f} :: \mathsf{a} \to \mathsf{b}) \; (\mathsf{xs} :: [\mathsf{a}]) \to \\ \textbf{case} \; \mathsf{xs} \; \textbf{of} \\ \mathsf{Nil} \; \langle \mathsf{a} \rangle & \to \mathsf{Nil} \; \langle \mathsf{b} \rangle \\ \mathsf{Cons} \; \langle \mathsf{a} \rangle \; \mathsf{y} \; \mathsf{ys} \to \mathsf{Cons} \; \langle \mathsf{b} \rangle \; (\mathsf{f} \; \mathsf{y}) \; (\mathsf{map} \; \langle \mathsf{a} \rangle \; \langle \mathsf{b} \rangle \; \mathsf{f} \; \mathsf{ys}) \end{array} \right.$$

Kind and type checking

Despite the generality of F_{ω} , checking kinds and types is easy compared to Haskell, because:

- nothing has to be inferred,
- ▶ the places of type abstraction and application are explicit.

Faculty of Science

Kind and type checking

Despite the generality of F_{ω} , checking kinds and types is easy compared to Haskell, because:

- nothing has to be inferred,
- ▶ the places of type abstraction and application are explicit.

Haskell's Hindley-Milner type system, on the other hand:

- implicitly generalizes functions on let bindings,
- implicitly instantiates polymorphic functions when applied.

Damas-Milner type inference is like filling in the missing type abstractions and type applications.

Kind rules

The kind system of F_{ω} is the simply-typed lambda calculus lifted to the level of types.

Variables:

$$\frac{\mathsf{a} :: \mathsf{K} \in \Gamma}{\Gamma \vdash \mathsf{a} :: \mathsf{K}}$$

Abstraction:

$$\frac{\Gamma, \mathsf{a} :: \mathsf{K}_1 \vdash \mathsf{t}_2 :: \mathsf{K}_2}{\Gamma \vdash \Lambda(\mathsf{a} :: \mathsf{K}_1) \to \mathsf{t}_2 :: \mathsf{K}_1 \to \mathsf{K}_2}$$

Kind rules (contd.)

Application:

$$\frac{\Gamma \vdash \mathsf{t}_1 :: \mathsf{K}_1 \to \mathsf{K}_2 \quad \Gamma \vdash \mathsf{t}_2 :: \mathsf{K}_1}{\Gamma \vdash (\mathsf{t}_1 \ \mathsf{t}_2) :: \mathsf{K}_2}$$

Function:

$$\frac{\Gamma \vdash \mathsf{t}_1 :: * \quad \Gamma \vdash \mathsf{t}_2 :: *}{\Gamma \vdash \mathsf{t}_1 \to \mathsf{t}_2 :: *}$$

Quantification:

$$\frac{\Gamma, \mathsf{a} :: \mathsf{K} \vdash \mathsf{t} :: *}{\Gamma \vdash \forall (\mathsf{a} :: \mathsf{K}).\mathsf{t} :: *}$$

Type rules

Variables:

$$\frac{x::t\in\Gamma}{\Gamma\vdash x::t}$$

Abstraction:

$$\frac{\Gamma \vdash \mathsf{t}_1 :: * \quad \Gamma, \mathsf{x} :: \mathsf{t}_1 \vdash \mathsf{e} :: \mathsf{t}_2}{\Gamma \vdash \lambda(\mathsf{x} :: \mathsf{t}_1) \rightarrow \mathsf{e} :: \mathsf{t}_1 \rightarrow \mathsf{t}_2}$$

Application:

$$\frac{\Gamma \vdash \mathsf{e}_1 :: \mathsf{t}_1 \to \mathsf{t}_2 \quad \Gamma \vdash \mathsf{e}_2 :: \mathsf{t}_1}{\Gamma \vdash (\mathsf{e}_1 \ \mathsf{e}_2) :: \mathsf{t}_2}$$

Type rules (contd.)

Type abstraction:

$$\frac{\Gamma, \mathsf{a} :: \mathsf{K} \vdash \mathsf{e} :: \mathsf{t}}{\Gamma \vdash \Lambda(\mathsf{a} :: \mathsf{K}) \rightarrow \mathsf{e} :: \forall (\mathsf{a} :: \mathsf{K}).\mathsf{t}}$$

Type application:

$$\frac{\Gamma \vdash \mathsf{e} :: \forall (\mathsf{a} :: \mathsf{K}).\mathsf{t}_2 \quad \Gamma \vdash \mathsf{t}_1 :: \mathsf{K}}{\Gamma \vdash (\mathsf{e} \ \langle \mathsf{t}_1 \rangle) :: \mathsf{t}_2 \{\mathsf{a} \mapsto \mathsf{t}_1\}}$$

A simple example

$$\underbrace{\frac{\mathbf{a} :: * \in \mathbf{a} :: *}{\mathbf{a} :: * \vdash \mathbf{a} :: *}}_{\mathbf{a} :: * \vdash \mathbf{a} :: *} \underbrace{\frac{\mathbf{x} :: \mathbf{a} \in \mathbf{a} :: *, \mathbf{x} :: \mathbf{a}}{\mathbf{a} :: *, \mathbf{x} :: \mathbf{a} \vdash \mathbf{x} :: \mathbf{a}}}_{\mathbf{a} :: * \vdash \lambda(\mathbf{x} :: \mathbf{a}) \to \mathbf{x} :: \mathbf{a} \to \mathbf{a}}$$

$$\emptyset \vdash \Lambda(\mathbf{a} :: *) \to \lambda(\mathbf{x} :: \mathbf{a}) \to \mathbf{x} :: \forall (\mathbf{a} :: *).\mathbf{a} \to \mathbf{a}$$

A simple example

$$\frac{\underbrace{a :: * \in a :: *}_{a :: * \vdash a :: *} \quad \underbrace{x :: a \in a :: *, x :: a}_{a :: *, x :: a \vdash x :: a}}_{a :: * \vdash \lambda(x :: a) \rightarrow x :: a \rightarrow a}$$

$$\emptyset \vdash \Lambda(a :: *) \rightarrow \lambda(x :: a) \rightarrow x :: \forall (a :: *).a \rightarrow a$$

- Type checking is completely syntax-directed and therefore simple: at any point, its clear which rule to apply.
- ▶ A consequence is that every F_{ω} term has either no or exactly one type.

Haskell vs. F_{ω}

- ▶ Quite a few extensions are needed to move Haskell toward the full power of F_{ω} : higher-rank types, impredicativity, scoped type variables.
- \blacktriangleright Even so, F_{ω} allows lambda on the type level everywhere, whereas Haskell has a very restricted form of type synonyms.
- Nevertheless, F_{ω} is clearly not feasible for programming, there are far too many annotations required.

14.3 From F_{ω} to GHC's core



Extra features

- ightharpoonup F_{ω} does not have any form of data types, pattern matching and type classes.
- ► GHC adds data types with simple pattern matching to the core. Complex pattern matching is translated into simple patterns, type classes are translated using evidence and dictionaries (both discussed before and in assignments).
- ▶ The constraints in GADTs as well as in type families are added on top of F_{ω} in the form of primitive type equality constraints with their own syntax and type rules.

Faculty of Science

Viewing GHC's core language

- ▶ By passing -fext-core, you can make GHC dump the translation of your Haskell modules in core language to a .hcr file.
- ► Furthermore, there are various debug flags for GHC that make GHC dump (pieces of) your program in core language as well. Unfortunately, the syntax of these outputs isn't entirely normalized.
- Knowing how to read core can be invaluable in debugging Haskell code, and learning what kind of optimizations GHC performs.

Faculty of Science

Viewing GHC's core language (contd.)

The program

```
module Id where  \{ -\# \text{ NOINLINE id } \# - \}  id x = x test = Id.id False
```

is translated to:

Observations

- ► The core output is difficult to read because of all the long names. Every identifier is fully qualified with package name and module.
- Symbols and internal names are encoded using z-encoding. If you see the letter 'z' occurring, it typically encodes a symbol. For instance, 'zi' encodes a period, and 'zm' a minus.
- Type abstraction and application is indicated using the '@' symbol.
- ► Recursive groups are indicated using %rec.
- Generally, % indicates special constructs of the core language.



[Faculty of Science

Another core example

```
module Eq where
test = Nothing == Just 'c'
```

```
%module main:Eq
 %rec
 {zddEqrgq :: (base:GHCziClasses.ZCTEq
                ((base:DataziMaybe.Maybe ghczmprim:GHCziTypes.Char))) =
     base:DataziMaybe.zdf3 @ ghczmprim:GHCziTypes.Char
     base:GHCziBase.zdf4:
  main:Eq.test :: ghczmprim:GHCziBool.Bool =
     base: GHCziClasses.zeze
     @ ((base:DataziMaybe.Maybe ghczmprim:GHCziTypes.Char)) zddEqrgq
     (base:DataziMaybe.Nothing @ ghczmprim:GHCziTypes.Char)
     (base:DataziMaybe.Just @ ghczmprim:GHCziTypes.Char
      (ghczmprim:GHCziTypes.Czh ('c'::ghczmprim:GHCziPrim.Charzh)))};
```

