uulogo.pdf

[Faculty of **Science**
Information and Computing Sciences]

Advanced Functional Programming

2012-2013, periode 2

c.jpg

Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

Nov 29, 2012

7. More Types, Lambda Cube

uulogo.pdf

This lecture

uulogo.pdf

7.1 The Equality type

uulogo.pdf

Comparing the length of vectors

```
equalLength :: Vec a m → Vec b n → Bool
```

uulogo.pdf

Comparing the length of vectors

```
equalLength :: Vec a m → Vec b n → Bool
```

Not useful, because

```
if equalLength v w then head (zipWith (,) v w)
                   else ...
```

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

uulogo.pdf

Equality type

```
data Equal :: * → * → * where
  Refl :: Equal a a
```

uulogo.pdf

Equality type

```
data Equal :: * → * → * where
  Refl :: Equal a a

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

uulogo.pdf

Equality type

```
data Equal :: * → * → * where
  Refl :: Equal a a

equalLength :: Vec a m → Vec b n → Maybe (Equal m n)
equalLength Nil          Nil      = Just Refl
equalLength (Cons x xs) (Cons y ys) =
  case equalLength xs ys of
    Just Refl                     → Just Refl
    Nothing                       → Nothing
equalLength _            _        = Nothing

test :: Vec a m → Vec b (Succ n) → (a, b)
test v w = case equalLength v w of
             Just Refl    → head (zipWith (,) v w)
```

uulogo.pdf

Expressive power of equality

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

```
data Expr :: * → * where
  Int  :: Int                                  → Expr Int
  If   :: Expr Bool → Expr a → Expr a          → Expr a
  Pair :: Expr a    → Expr b                   → Expr (a, b)
  Fst  :: Expr (a, b)                          → Expr a
    ...
data Expr t =
        Int  (Equal t Int)     Int
  |     If                     (Expr Bool) (Expr t) (Expr t)
  | ∀a b.Pair (Equal t (a, b)) (Expr a) (Expr b)
  | ∀a b.Fst  (Equal t a)      (Expr (a, b))
  | ...
```

uulogo.pdf

Outlook generic programming: Reflecting types

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

We can now write generic functions as functions of the form

```
f :: Type a → ... a ...
```

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

```
data Dynamic :: * where
  Dyn :: Type a → a → Dynamic
```

uulogo.pdf

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 $\bot$:

  > $\bot$ :: Equal Int Bool

uulogo.pdf

7.2 Higher-rank polymorphism

uulogo.pdf

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                  → ST s (STRef s a)
readSTRef  :: STRef s a         → ST s a
writeSTRef :: STRef s a → a     → ST s ()

runST :: (∀s.ST s a) → a
```

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

uulogo.pdf

Preventing escaping references

This should fail:

$$
\begin{aligned}
&t_1 :: \mathsf{ST}\ s_1\ (\mathsf{STRef}\ s_1\ \mathsf{Int}) \\
&t_1 = \mathsf{newSTRef}\ 0 \\
&t_2 :: \mathsf{STRef}\ s_2\ \mathsf{Int} \to \mathsf{ST}\ s_2\ \mathsf{Int} \\
&t_2\ ref_2 = \mathsf{readSTRef}\ ref_2 \\
&t_3 :: \mathsf{Int} \\
&t_3 = \mathsf{runST}\ (t_2\ (\mathsf{runST}\ t_1))
\end{aligned}
$$

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

uulogo.pdf

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.

uulogo.pdf

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:

> runST :: $(\forall s.ST\ s\ a) \rightarrow a$

uulogo.pdf

Why does it fail (contd.)

$$
\begin{aligned}
&t_1 :: \mathsf{ST}\ s_1\ (\mathsf{STRef}\ s_1\ \mathsf{Int}) \\
&t_1 = \mathsf{newSTRef}\ 0 \\
&t_2 :: \mathsf{STRef}\ s_2\ \mathsf{Int} \to \mathsf{ST}\ s_2\ \mathsf{Int} \\
&t_2\ \mathsf{ref}_2 = \mathsf{readSTRef}\ \mathsf{ref}_2 \\
&t_3 :: \mathsf{Int} \\
&t_3 = \mathsf{runST}\ (t_2\ (\mathsf{runST}\ t_1))
\end{aligned}
$$

- The function t_1 is polymorphic in s_1.
- When run, t_1 is instantiated at a particular type, let's call it tag.
- The result of t_1 is thus STRef tag Int, and not independent of tag.

uulogo.pdf

7.3 The rank of a type

uulogo.pdf

Rank-1 polymorphism

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

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

  is an abbreviation for

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

uulogo.pdf

Rank-n polymorphism

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

  > runST :: $\forall$a.($\forall$s.ST s a) $\rightarrow$ 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 $\forall$ is not optional.

uulogo.pdf

The difference

Compare:

$$\begin{aligned} &\mathsf{runST} :: \forall a.(\forall s.\mathsf{ST}\ s\ a) \to a \\ &\mathsf{runST}' :: \forall a\ s.\quad \mathsf{ST}\ s\ a \to a \end{aligned}$$

uulogo.pdf

The difference

Compare:

$$\begin{aligned} &\mathsf{runST} :: \forall a.(\forall s.\mathsf{ST}\ s\ a) \to a \\ &\mathsf{runST}' :: \forall a\ s.\quad \mathsf{ST}\ s\ a \to a \end{aligned}$$

- 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.

uulogo.pdf

Other uses

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

uulogo.pdf

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.

uulogo.pdf

7.4 Impredicativity

uulogo.pdf

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?

uulogo.pdf

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**.

uulogo.pdf

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`).

uulogo.pdf

Data structures containing polymorphic values

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

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

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

uulogo.pdf

8. Systems F, F_ω, FC

uulogo.pdf

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.

uulogo.pdf

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.

uulogo.pdf

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).

uulogo.pdf

8.1 The lambda cube

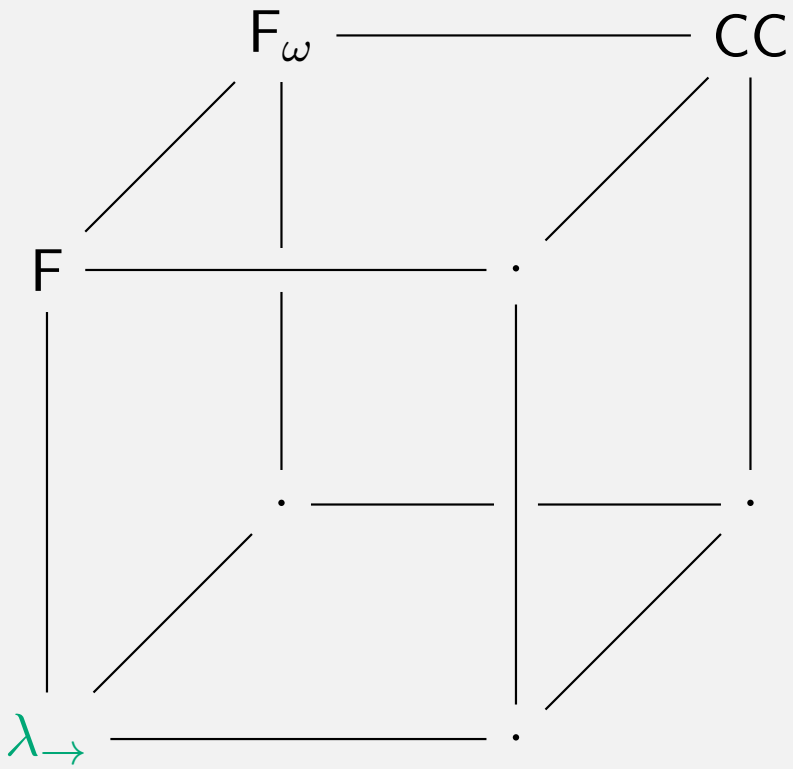
uulogo.pdf

The lambda cube

- 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)

uulogo.pdf

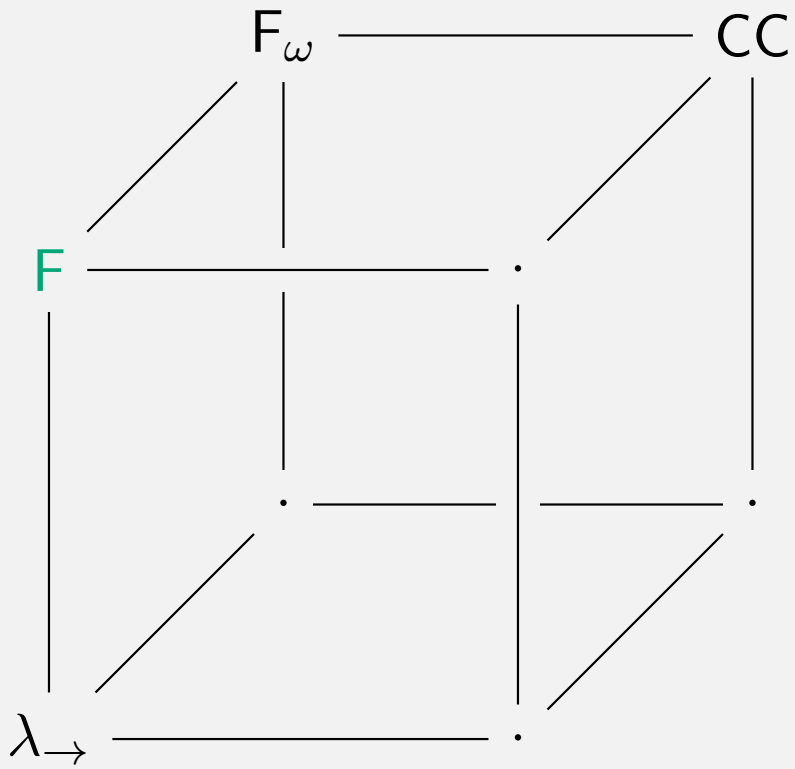
Important points in the lambda cube



$\lambda_\rightarrow$ – the **simply typed lambda calculus** has no polymorphism and no kind system

uulogo.pdf

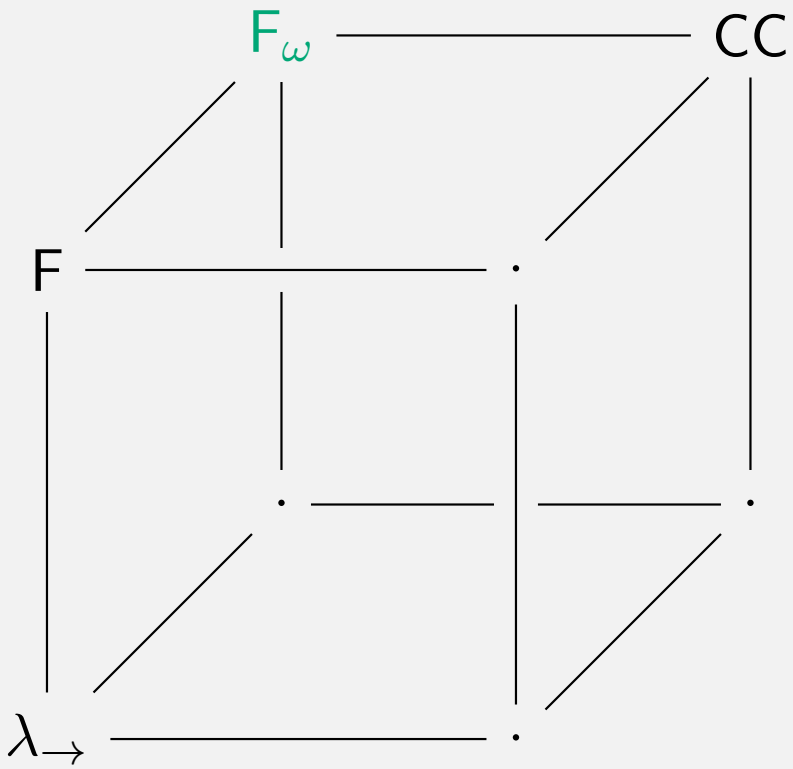
Important points in the lambda cube



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

uulogo.pdf

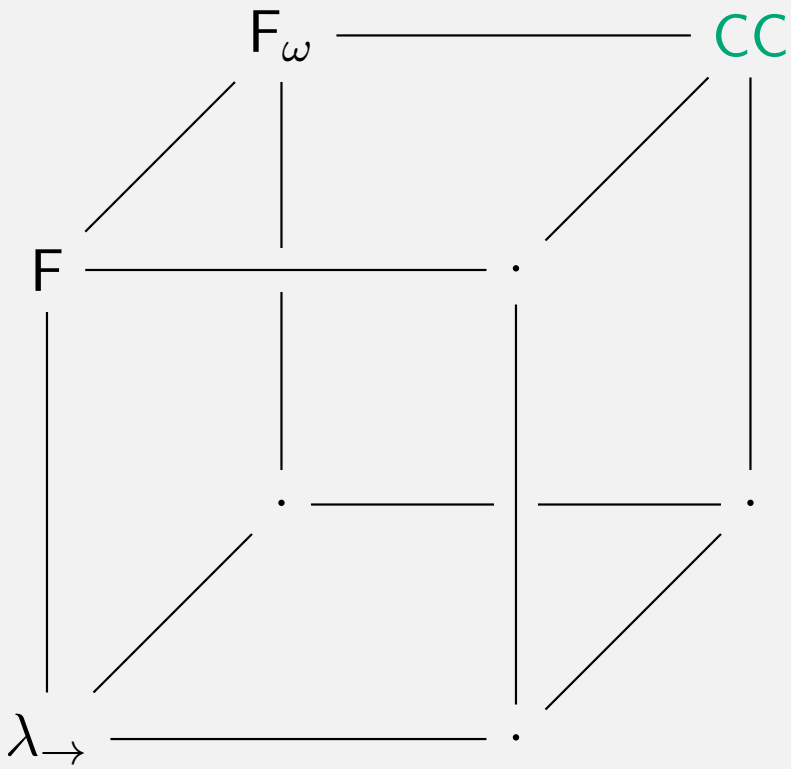
Important points in the lambda cube



F_ω – **System F_ω** has polymorphism and kinds

uulogo.pdf

Important points in the lambda cube



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

uulogo.pdf

8.2 F_ω

uulogo.pdf

Syntax

Expressions:

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

Note how all lambda abstractions are explicitly typed.

uulogo.pdf

Syntax (contd.)

Types:

```
t ::= a                    -- type variable
    | t → t                -- function type
    | ∀(a :: K).t          -- universal type
    | Λ(a :: K) → t        -- type constructors
    | (t t)                -- application
```

Kinds:

```
K ::= ∗          -- base kind
    | K → K      -- function kind
```

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

uulogo.pdf

Syntax examples

Polymorphic identity:

$$\Lambda(a :: *) \to \lambda(x :: a) \to x$$

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

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

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

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

uulogo.pdf

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.

uulogo.pdf

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.

uulogo.pdf

Kind rules

The kind system of F_ω is the simply-typed lambda calculus lifted to the level of types.

Variables:

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

Abstraction:

$$\frac{\Gamma, a :: K_1 \vdash t_2 :: K_2}{\Gamma \vdash \Lambda(a :: K_1) \to t_2 :: K_1 \to K_2}$$

uulogo.pdf

Kind rules (contd.)

Application:

$$\frac{\Gamma \vdash t_1 :: K_1 \to K_2 \quad \Gamma \vdash t_2 :: K_1}{\Gamma \vdash (t_1\ t_2) :: K_2}$$

Function:

$$\frac{\Gamma \vdash t_1 :: * \quad \Gamma \vdash t_2 :: *}{\Gamma \vdash t_1 \to t_2 :: *}$$

Quantification:

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

uulogo.pdf

Type rules

Variables:

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

Abstraction:

$$\frac{\Gamma \vdash t_1 :: * \quad \Gamma, x :: t_1 \vdash e :: t_2}{\Gamma \vdash \lambda(x :: t_1) \to e :: t_1 \to t_2}$$

Application:

$$\frac{\Gamma \vdash e_1 :: t_1 \to t_2 \quad \Gamma \vdash e_2 :: t_1}{\Gamma \vdash (e_1\ e_2) :: t_2}$$

uulogo.pdf

Type rules (contd.)

Type abstraction:

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

Type application:

$$\frac{\Gamma \vdash e :: \forall(a :: K).t_2 \quad \Gamma \vdash t_1 :: K}{\Gamma \vdash (e\ \langle t_1 \rangle) :: t_2\{a \mapsto t_1\}}$$

uulogo.pdf

A simple example

$$\dfrac{\dfrac{\dfrac{a :: * \in a :: *}{a :: * \vdash a :: *} \quad \dfrac{x :: a \in a :: *, x :: a}{a :: *, x :: a \vdash x :: a}}{a :: * \vdash \lambda(x :: a) \to x :: a \to a}}{\emptyset \vdash \Lambda(a :: *) \to \lambda(x :: a) \to x :: \forall(a :: *).a \to a}$$

uulogo.pdf

A simple example

$$\dfrac{\dfrac{\dfrac{a :: * \in a :: *}{a :: * \vdash a :: *} \quad \dfrac{x :: a \in a :: *, x :: a}{a :: *, x :: a \vdash x :: a}}{a :: * \vdash \lambda(x :: a) \to x :: a \to a}}{\emptyset \vdash \Lambda(a :: *) \to \lambda(x :: a) \to x :: \forall(a :: *).a \to a}$$

- 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.

uulogo.pdf

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.
- 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.

uulogo.pdf

8.3 From F_ω to GHC's core

uulogo.pdf

Extra features

- 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.

uulogo.pdf

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.

`uulogo.pdf`

Viewing GHC's core language (contd.)

The program

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

is translated to:

```
%module main:Id
   %rec
   {main:Id.id :: %forall tafu . tafu -> tafu =
      \ @ tafu (xafr::tafu) -> xafr;
    main:Id.test :: ghczmprim:GHCziBool.Bool =
      main:Id.id @ ghczmprim:GHCziBool.Bool ghczmprim:GHCziBool.False};
```

uulogo.pdf

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.

`uulogo.pdf`

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)))};
```

uulogo.pdf

Still problematic

```
f :: ∀a.Int → a → (exists a ∘ (a, a → Int → Int))
let (a, a2I2I) = f 3 a
in a2I2I 4 a
```

uulogo.pdf

Still problematic

```
f :: ∀a.Int → a → (exists a ∘ (a, a → Int → Int))
let (a, a2I2I) = f 3 a
in a2I2I 4 a
```

For lazy evaluation at the value level we may need evaluation at the type level.

uulogo.pdf

wxHaskell: GUI programming

- We will become a bit more practical after all this heavy theory.
- Get wxHaskell and reactive – banana.

uulogo.pdf