Slide 1

Core in LF

Simon Winwood
University of New South Wales
sjw@cse.unsw.edu.au

① The Core Intermediate Language
② System Fω and Core's Semantics
③ The Logical Framework

Slide 2

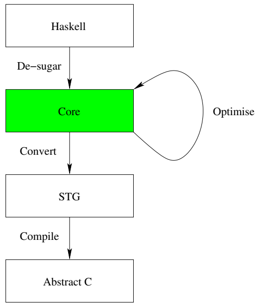
MOTIVATION

A machine-checked semantics for Core

➜ Type soundness
➜ Correctness of transformations (optimisations)
➜ Correctness of compilation

Slide 3

THE CORE IL



➜ Allow as many optimisations on Core as possible

Slide 4

Features of Core:

Core is desugared Haskell

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = (f x) : (map f xs)
```

becomes

```
map = /\ a b ->
      \f::(a -> b) xs::List a ->
         case xs of
              Nil       -> Nil
              Cons y ys -> Cons (f y) (map f ys)
```

Slide 5

Haskell 1A:

- Lazy semantics
  - Call-by-name (call-by-need).
- Higher-order functions
  - Type system with ∀, type abstraction.
- Higher-order types
  - Type system with type-level application.

Core is (sortof) System Fω extended with `let, case`, constructors, and primitives.

Slide 6

Representation of Core Expressions:

```
let
   x:Int = fac 100
in
   case (x + x) of
       y -> y
```

(whiteboard)

Slide 7

Unboxed values:

Boxed arithmetic is slow!

Consider `a + b`

```
boxed_int builtin_plus(boxed_int a, boxed_int b)
{
    int tmpa;
    int tmpb;

    tmpa = FORCE(a);
    tmpb = FORCE(b);

    return BOX(tmpa + tmpb);
}
```

Slide 8

Solution:

Don't box values!

In Core: allow values of unboxed type.

- introduce unboxed types (`Int#, Float#`, etc.)
- `data Int = I# Int#`
- allows/requires usage of strictness analysis

```
+ = \a b ->
    case a of
         I# a# -> case b of
                I# b# -> case (a# +# b#) of
                       r# -> I# r#
```

Slide 9

Example `(x + y) * z`:

Inline *:

```
case (x + y) of
     I# a# -> case z of
                   I# b# -> case (a# *# b#) of
                             r# -> I# r#
```

Slide 10

Example `(x + y) * z` (cont.):

Inline +:

```
case (case x of
         I# a# -> case y of
               I# b# -> case (a# +# b#) of
                     r# -> I# r#)
     of
     I# c# -> case z of
                   I# d# -> case (c# *# d#) of
                             f# -> I# f#
```


Slide 11

Example `(x + y) * z` (cont.):

Case of known value:

```
case x of
 I# a# -> case y of
            I# b# -> case (a# +# b#) of
                      r# -> case z of
                             I# d# -> case (r# *# d#) of
                                       f# -> I# f#
```

No intermediate boxed value!

Slide 12

BUT DOESN'T THIS BREAK TYPE SAFETY/LAZINESS/SOMETHING?

Type Theory of Core

But first: $\lambda_\rightarrow$: Syntax:

Slide 13

$$\begin{array}{rcl} \tau & ::= & c \mid \sigma \rightarrow \tau \\ \\ e & ::= & \lambda x : \tau . e \\ & | & e_1 e_2 \\ & | & k \end{array}$$

$\lambda_\rightarrow$: Static Semantics:

Slide 14

$$(\mathrm{Var})\frac{\mathrm{x} : \tau \in \Gamma}{\Gamma \vdash \mathrm{x} : \tau}$$

$$(\mathrm{Abs})\frac{\Gamma, \mathrm{x} : \tau_1 \vdash \mathrm{e} : \tau_2}{\Gamma \vdash \lambda \mathrm{x} : \tau_1 . \mathrm{e} : \tau_1 \rightarrow \tau_2} \quad (\mathrm{App})\frac{\Gamma \vdash \mathrm{e}_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash \mathrm{e}_2 : \tau_1}{\Gamma \vdash \mathrm{e}_1 \mathrm{e}_2 : \tau_2}$$

(Curry-Howard) Note similarity to propositional logic: (Var) = Ax, (Abs) = deduction, (App) = MP.

$\lambda_\rightarrow$: (Lazy) Operational Semantics:

Slide 15

$$(\mathrm{App})\frac{\mathrm{e}_1 \longrightarrow \mathrm{e}_1'}{\mathrm{e}_1 \mathrm{e}_2 \longrightarrow \mathrm{e}_1' \mathrm{e}_2}$$

$$(\beta - \mathrm{reduction})\frac{}{(\lambda \mathrm{x}.\mathrm{e}_1)\mathrm{e}_2 \longrightarrow [\mathrm{x} \mapsto \mathrm{e}_2]\mathrm{e}_1}$$

$$(\mathrm{Prim})\frac{}{\mathrm{ke} \longrightarrow \delta(\mathrm{k}, \mathrm{e})}$$

Type safety:

Slide 16

Theorem (Progress). *If $e : \tau$ then, for some e' $e \longrightarrow e'$.*

Proof. By induction on $e : \tau$. □

Theorem (Preservation). *If $e : \tau$ then and $e \longrightarrow e'$ then $e' : \tau$.*

Proof. By induction on $e : \tau$. □

Lemma (Inversion). *Inversion of the typing relation:*

1. *If $\Gamma \vdash x : \tau$ then $x : \tau \in \Gamma$*
2. *If $\Gamma \vdash \lambda x : \tau_1 . e : \tau$ then $\tau = \tau_1 \rightarrow \sigma$ and $\Gamma, x : \tau_1 \vdash e : \sigma$*
3. *...*

Lemma (Substitution). *If $\Gamma, x : \tau_1 \vdash e_1 : \tau_2$ and $\Gamma \vdash e_2 : \tau_1$ then $\Gamma \vdash [x \mapsto e_2]e_1 : \tau_2$.*

Slide 17

System F: Syntax:

$\lambda_{\rightarrow}$ + polymorphic functions

$$
\begin{array}{rcl}
\tau & ::= & c \mid \sigma \rightarrow \tau \\
& \mid & \alpha \mid \forall \alpha . \tau \\
\\
e & ::= & \lambda x : \tau . e \\
& \mid & e_1 e_2 \\
& \mid & \Lambda \alpha . e \\
& \mid & e[\tau] \\
& \mid & k
\end{array}
$$

Slide 18

System F: Semantics:

$$
(\mathrm{Var})\frac{\mathrm{x} : \tau \in \Gamma}{\Gamma \vdash \mathrm{x} : \tau}
$$

$$
(\mathrm{Abs})\frac{\Gamma, \mathrm{x} : \tau_1 \vdash \mathrm{e} : \tau_2}{\Gamma \vdash \lambda \mathrm{x} : \tau_1 . \mathrm{e} : \tau_1 \rightarrow \tau_2} \quad (\mathrm{App})\frac{\Gamma \vdash \mathrm{e}_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash \mathrm{e}_2 : \tau_1}{\Gamma \vdash \mathrm{e}_1 \mathrm{e}_2 : \tau_2}
$$

$$
(\mathrm{TAbs})\frac{\Gamma, \alpha \vdash \mathrm{e} : \tau}{\Gamma \vdash \Lambda \alpha . \mathrm{e} : \forall \alpha . \tau} \qquad (\mathrm{TApp})\frac{\Gamma \vdash \mathrm{e} : \forall \alpha . \tau}{\Gamma \vdash \mathrm{e}[\mathrm{T}] : [\alpha \mapsto \mathrm{T}]\tau}
$$

Slide 19

What about higher-order types?:

For example:

```
data Pair a b = ...
```

System Fω allows type abstraction and application. But how to ensure that types are well formed?

Kinds! A type-system for types.

For example

`Pair:`$(* \rightarrow * \rightarrow *)$ = $\lambda\alpha\lambda\beta\forall\gamma.(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \gamma$

`Pair Int Int:(*)` = $\forall\gamma.(\mathtt{Int} \rightarrow \mathtt{Int} \rightarrow \gamma) \rightarrow \gamma$

Note: $\forall \alpha . \tau$ is different to $\lambda \alpha . \tau$!

Slide 20

System Fω: Syntax:

System F + higher-order types.

$$
\begin{array}{rcl}
\kappa & ::= & * \mid \kappa \rightarrow \kappa \\
\tau & ::= & c \mid \sigma \rightarrow \tau (or \rightarrow :: (* \rightarrow * \rightarrow *)) \\
& \mid & \alpha \mid \forall \alpha :: \kappa . \tau \\
& \mid & \lambda \alpha :: \kappa . \tau \mid \tau_1 \tau_2 \\
e & ::= & \lambda x : \tau . e \\
& \mid & e_1 e_2 \\
& \mid & \Lambda \alpha :: \kappa . e \\
& \mid & e[\tau] \\
& \mid & k
\end{array}
$$

System Fω: Semantics:

$$(\mathrm{Var})\frac{\mathrm{x} : \tau \in \Gamma}{\Gamma \vdash \mathrm{x} : \tau}$$

$$(\mathrm{Abs})\frac{\Gamma, \mathrm{x} : \tau_1 \vdash \mathrm{e} : \tau_2}{\Gamma \vdash \lambda \mathrm{x} : \tau_1 . \mathrm{e} : \tau_1 \to \tau_2} \quad (\mathrm{App})\frac{\Gamma \vdash \mathrm{e}_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash \mathrm{e}_2 : \tau_1}{\Gamma \vdash \mathrm{e}_1 \mathrm{e}_2 : \tau_2}$$

$$(\mathrm{TAbs})\frac{\Gamma, \alpha :: \kappa \vdash \mathrm{e} : \tau}{\Gamma \vdash \Lambda\alpha :: \kappa . \mathrm{e} : \forall \alpha :: \kappa . \tau} \quad (\mathrm{TApp})\frac{\Gamma \vdash \mathrm{e} : \forall \alpha :: \kappa . \tau \quad \Gamma \vdash \mathrm{T} :: \kappa}{\Gamma \vdash \mathrm{e}[\mathrm{T}] : [\alpha \mapsto \mathrm{T}]\tau}$$

$$(\mathrm{Eq})\frac{\Gamma \vdash \mathrm{e} : \tau \quad \tau \equiv \sigma \quad \Gamma \vdash \sigma :: *}{\Gamma \vdash \mathrm{e} : \sigma}$$

Slide 21

Aside: Problem with Eq:

Definitional type equality $\equiv$ is refl, symm, trans closure of β-conversion (plus rules for $\forall$, $\to$, etc.)

When are two types equal?

$\mathtt{Nat} \to (\lambda\alpha :: *.\alpha)\mathtt{Nat}$

$\mathtt{Nat} \to \mathtt{Nat}$

Abstraction and application at type level means that types are no longer unique.

Inversion no longer holds!

Slide 22

What about Core?:

Somewhere between System F and System Fω.

- ➜ No abstraction (no anonymous types)
- ➜ Application replaced by constructor instantiation (i.e., still need kinds)

$$\begin{array}{lcl}
\kappa & ::= & * \mid \# \mid ? \mid \kappa \to \kappa \\
\sigma & ::= & \forall \alpha . \sigma \mid \tau \\
\tau & ::= & \pi \mid \nu \\
\pi & ::= & \alpha \mid \tau_1 \to \tau_2 \\
 & \mid & \chi\ \pi_1 \ldots \pi_n \\
\nu & ::= & Int\# \mid \ldots \\
 & \mid & \chi\#\ \pi_1\ \ldots\ \pi_n
\end{array}$$

Slide 23

Core syntax (cont.):

$$\begin{array}{lcl}
e & ::= & \lambda x : \tau . e \\
 & \mid & e_1 e_2 \\
 & \mid & \Lambda\alpha :: \kappa . e \\
 & \mid & e[\tau] \\
 & \mid & \mathtt{let}\ x : \tau\ =\ e_1\ \mathtt{in}\ e_2 \\
 & \mid & \mathtt{case}\ e\ \mathtt{of}\ a_1\ v_1^1 \ldots v_1^n \mathtt{->}\ e_1; \ldots \\
 & \mid & k
\end{array}$$

Note: `let` *builds* closures, `case` *evaluates* closures.

Slide 24

Slide 25

Core: Static Semantics:

$$(\mathrm{Var})\frac{\mathrm{x} : \tau \in \Gamma}{\Gamma \vdash \mathrm{x} : \tau} \qquad (\mathrm{Let})\frac{\Gamma \vdash \mathrm{e}_1 : \tau_1 \quad \Gamma, \mathrm{x} : \tau_1 \vdash \mathrm{e}_2 : \tau_2}{\Gamma \vdash \texttt{let x = } \mathrm{e}_1 \texttt{ in } \mathrm{e}_2 : \tau_2}$$

$$(\mathrm{Abs})\frac{\Gamma, \mathrm{x} : \tau_1 \vdash \mathrm{e} : \tau_2}{\Gamma \vdash \lambda \mathrm{x} : \tau_1 . \mathrm{e} : \tau_1 \to \tau_2} \quad (\mathrm{App})\frac{\Gamma \vdash \mathrm{e}_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash \mathrm{e}_2 : \tau_1}{\Gamma \vdash \mathrm{e}_1 \mathrm{e}_2 : \tau_2}$$

$$(\mathrm{TAbs})\frac{\Gamma, \alpha :: \kappa \vdash \mathrm{e} : \tau}{\Gamma \vdash \Lambda \alpha :: \kappa . \mathrm{e} : \forall \alpha :: \kappa . \tau} \quad (\mathrm{TApp})\frac{\Gamma \vdash \mathrm{e} : \forall \alpha :: \kappa . \tau \quad \Gamma \vdash \mathrm{T} :: \kappa \quad \kappa \neq \{\#, ?\}}{\Gamma \vdash \mathrm{e}[\mathrm{T}] : [\alpha \mapsto \mathrm{T}]\tau}$$

$$(\mathrm{Case})\frac{\Gamma \vdash \mathrm{e} : \tau_1 \quad \Gamma \vdash \mathrm{a_i} :: \tau_1 \quad \Gamma, \mathrm{v_i^1} : \sigma_1, \ldots, \mathrm{v_i^n} : \sigma_n \vdash \mathrm{e_i} : \tau_2}{\Gamma \vdash \texttt{case } \mathrm{e} \texttt{ of } \mathrm{a}_1\ \mathrm{v}_1^1 \ldots \mathrm{v}_1^n \texttt{->}\ \mathrm{e}_1; \ldots : \tau_2}$$

$$(\mathrm{App(kinds)})\frac{\Gamma \vdash \mathrm{T}_1 :: (\kappa_1 \to \kappa_2) \quad \Gamma \vdash \mathrm{T}_2 :: \kappa_1}{\Gamma \vdash \mathrm{T}_1 \mathrm{T}_2 :: (\kappa_2)}$$

Slide 26

Core: Unboxed types:

No polymorphism over unboxed types

- Garbage collection
- Value size
- (separate compilation)

So ...

- # is kind of unboxed types
- Cannot unify type variable with type of kind #

Where do they come from?

- pre-defined constants `Int#`, etc.
  - machine-manipulatable types
- `data unboxed UPair a b = ...`
  - C-like struct of boxed types.

Slide 27

THE LOGICAL FRAMEWORK

The Logical Framework (LF) is (essentially) System F with dependent types and kinds:

$$\begin{array}{rcl} K & ::= & \texttt{type} \mid \Pi x : A.K \\ A & ::= & a \mid \Pi x : A.B \mid \lambda x : A.B \mid AM \\ M & ::= & c \mid x \mid \lambda x : A.M \mid MN \end{array}$$

Judgements-as-types: represent judgements (e.g. $\vdash e : \tau, e \longrightarrow e'$) as types (see example).

Proof witnesses are objects of type representing the proof.

Slide 28

An Example: $\lambda_\to$:

Slide 29

THANK YOU. QUESTIONS?