Programming Language Concepts: Lecture 19

Madhavan Mukund

Chennai Mathematical Institute

madhavan@cmi.ac.in

http://www.cmi.ac.in/~madhavan/courses/pl2009

PLC 2009, Lecture 19, 01 April 2009
Adding types to $\lambda$-calculus

- The basic $\lambda$-calculus is untyped
- The first functional programming language, LISP, was also untyped
- Modern languages such as Haskell, ML, . . . are strongly typed
- What is the theoretical foundation for such languages?
Types in functional programming

The structure of types in Haskell

- Basic types—\textit{Int}, \textit{Bool}, \textit{Float}, \textit{Char}
Types in functional programming

The structure of types in Haskell

- **Basic types**—`Int`, `Bool`, `Float`, `Char`

- **Structured types**
  
  - **Lists** If `a` is a type, so is `[a]`
  
  - **Tuples** If `a1`, `a2`, ..., `ak` are types, so is `(a1,a2,...,ak)`
Types in functional programming

The structure of types in Haskell

- Basic types—\texttt{Int}, \texttt{Bool}, \texttt{Float}, \texttt{Char}

- Structured types
  - \textbf{Lists} \ If \( a \) is a type, so is \([a]\)
  
  \textbf{Tuples} \ If \( a_1, a_2, \ldots, a_k \) are types, so is \((a_1,a_2,\ldots,a_k)\)

- Function types
  - If \( a, b \) are types, so is \( a \rightarrow b \)
  
  Function with input \( a \), output \( b \)
Types in functional programming

The structure of types in Haskell

- **Basic types**—Int, Bool, Float, Char

- **Structured types**
  - **[Lists]** If a is a type, so is [a]
  - **[Tuples]** If a₁, a₂, ..., aₖ are types, so is (a₁, a₂, ..., aₖ)

- **Function types**
  - If a, b are types, so is a → b
  - Function with input a, output b

- **User defined types**
  - Data day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
  - Data BTree a = Nil | Node (BTree a) a (Btree a)
Adding types to $\lambda$-calculus . . .

Set $\Lambda$ of untyped lambda expressions is given by

$$\Lambda = x \mid \lambda x. M \mid MM'$$

where $x \in \text{Var}$, $M, M' \in \Lambda$. 
Adding types to $\lambda$-calculus . . .

- Set $\Lambda$ of untyped lambda expressions is given by
  \[
  \Lambda = x \mid \lambda x. M \mid MM'
  \]
  where $x \in \text{Var}$, $M, M' \in \Lambda$.

- Add a syntax for basic types

- When constructing expressions, build up the type from the types of the parts
Adding types to $\lambda$-calculus . . .

- Restrict our language to have just one basic type, written as $\tau$
Adding types to $\lambda$-calculus . . .

- Restrict our language to have just one basic type, written as $\tau$
- No structured types (lists, tuples, . . .)
Adding types to $\lambda$-calculus . . .

- Restrict our language to have just one basic type, written as $\tau$
- No structured types (lists, tuples, . . .)
- Function types arise naturally ($\tau \to \tau$, $(\tau \to \tau) \to \tau \to \tau$, . . .)
“Simply typed” $\lambda$-calculus

A separate set of variables $V_{ars}$ for each type $s$
“Simply typed” $\lambda$-calculus

A separate set of variables $Var_s$ for each type $s$

Define $\Lambda_s$, expressions of type $s$, by mutual recursion
“Simply typed” $\lambda$-calculus

A separate set of variables $\text{Var}_s$ for each type $s$

Define $\Lambda_s$, expressions of type $s$, by mutual recursion

► For each type $s$, every variable $x \in \text{Var}_s$ is in $\Lambda_s$
A separate set of variables $\text{Var}_s$ for each type $s$

Define $\Lambda_s$, expressions of type $s$, by mutual recursion

- For each type $s$, every variable $x \in \text{Var}_s$ is in $\Lambda_s$
- If $M \in \Lambda_t$ and $x \in \text{Var}_s$ then $(\lambda x. M) \in \Lambda_{s \rightarrow t}$. 
“Simply typed” $\lambda$-calculus

A separate set of variables $\text{Var}_s$ for each type $s$

Define $\Lambda_s$, expressions of type $s$, by mutual recursion

- For each type $s$, every variable $x \in \text{Var}_s$ is in $\Lambda_s$
- If $M \in \Lambda_t$ and $x \in \text{Var}_s$ then $(\lambda x.M) \in \Lambda_{s\rightarrow t}$.
- If $M \in \Lambda_{s\rightarrow t}$ and $N \in \Lambda_s$ then $(MN) \in \Lambda_t$.
  - Note that application must be well typed
“Simply typed” $\lambda$-calculus

A separate set of variables $\text{Var}_s$ for each type $s$

Define $\Lambda_s$, expressions of type $s$, by mutual recursion

- For each type $s$, every variable $x \in \text{Var}_s$ is in $\Lambda_s$

- If $M \in \Lambda_t$ and $x \in \text{Var}_s$ then $(\lambda x. M) \in \Lambda_{s \rightarrow t}$.

- If $M \in \Lambda_{s \rightarrow t}$ and $N \in \Lambda_s$ then $(MN) \in \Lambda_t$.
  - Note that application must be well typed

$\beta$ rule as usual

- $(\lambda x. M)N \rightarrow_\beta M\{x \leftarrow N\}$
“Simply typed” $\lambda$-calculus

A separate set of variables $\text{Var}_s$ for each type $s$
Define $\Lambda_s$, expressions of type $s$, by mutual recursion

- For each type $s$, every variable $x \in \text{Var}_s$ is in $\Lambda_s$
- If $M \in \Lambda_t$ and $x \in \text{Var}_s$ then $(\lambda x. M) \in \Lambda_{s \rightarrow t}$.
- If $M \in \Lambda_{s \rightarrow t}$ and $N \in \Lambda_s$ then $(MN) \in \Lambda_t$.
  - Note that application must be well typed

$\beta$ rule as usual

- $(\lambda x. M)N \rightarrow_{\beta} M\{x \leftarrow N\}$
- We must have $\lambda x. M \in \Lambda_{s \rightarrow t}$ and $N \in \Lambda_s$ for some types $s, t$
“Simply typed” λ-calculus

A separate set of variables $\text{Var}_s$ for each type $s$

Define $\Lambda_s$, expressions of type $s$, by mutual recursion

- For each type $s$, every variable $x \in \text{Var}_s$ is in $\Lambda_s$
- If $M \in \Lambda_t$ and $x \in \text{Var}_s$ then $(\lambda x. M) \in \Lambda_{s \to t}$.
- If $M \in \Lambda_{s \to t}$ and $N \in \Lambda_s$ then $(MN) \in \Lambda_t$.
  - Note that application must be well typed

$\beta$ rule as usual

- $(\lambda x. M)N \rightarrow_\beta M\{x \leftarrow N\}$
- We must have $\lambda x. M \in \Lambda_{s \to t}$ and $N \in \Lambda_s$ for some types $s, t$
- Moreover, if $\lambda x. M \in \Lambda_{s \to t}$, then $x \in \text{Var}_s$, so $x$ and $N$ are compatible
“Simply typed” $\lambda$-calculus ... 

- Extend $\rightarrow_{\beta}$ to one-step reduction $\rightarrow$, as usual
“Simply typed” $\lambda$-calculus . . .

- Extend $\rightarrow_\beta$ to one-step reduction $\rightarrow$, as usual

- The reduction relation $\rightarrow^*$ is Church-Rosser
“Simply typed” $\lambda$-calculus . . .

- Extend $\beta$ to one-step reduction $\rightarrow$, as usual
- The reduction relation $\rightarrow^*$ is Church-Rosser
- In fact, $\rightarrow^*$ satisifies a much strong property
A $\lambda$-expression is

- normalizing if it has a normal form.
Strong normalization

A \( \lambda \)-expression is

- **normalizing** if it has a normal form.

- **strongly normalizing** if every reduction sequence leads to a normal form
A $\lambda$-expression is

- **normalizing** if it has a normal form.
- **strongly normalizing** if every reduction sequence leads to a normal form

Examples

- $(\lambda x.xx)(\lambda x.xx)$ is not normalizing
Strong normalization

A \( \lambda \)-expression is

- **normalizing** if it has a normal form.

- **strongly normalizing** if every reduction sequence leads to a normal form.

Examples

- \((\lambda x.xx)(\lambda x.xx)\) is not normalizing

- \((\lambda yz.z)((\lambda x.xx)(\lambda x.xx))\) is not strongly normalizing.
A λ-calculus is strongly normalizing if every term in the calculus is strongly normalizing.
A λ-calculus is strongly normalizing if every term in the calculus is strongly normalizing.

**Theorem**

*The simply typed λ-calculus is strongly normalizing*
A $\lambda$-calculus is strongly normalizing if every term in the calculus is strongly normalizing.

**Theorem**

*The simply typed $\lambda$-calculus is strongly normalizing*

**Proof intuition**

- Each $\beta$-reduction reduces the type complexity of the term
- Cannot have an infinite sequence of reductions
Type checking

- Syntax of simply typed $\lambda$-calculus permits only well-typed terms
Type checking

- Syntax of simply typed $\lambda$-calculus permits only well-typed terms
- Converse question; Given an arbitrary term, is it well-typed?
Type checking

- Syntax of simply typed λ-calculus permits only well-typed terms

- Converse question; Given an arbitrary term, is it well-typed?
  - For instance, we cannot assign a valid type to $ff\ldots$
  - $\ldots$ so $ff$ is not a valid expression in this calculus
Type checking

- Syntax of simply typed $\lambda$-calculus permits only well-typed terms

- Converse question; Given an arbitrary term, is it well-typed?
  - For instance, we cannot assign a valid type to $f \ f \ldots$
  - $\ldots$ so $f \ f$ is not a valid expression in this calculus

**Theorem**

_The type-checking problem for the simply typed $\lambda$-calculus is decidable_
A term may admit multiple types

- \( \lambda x.x \) can be of type \( \tau \to \tau \), \( (\tau \to \tau) \to (\tau \to \tau) \), \ldots
Type checking . . .

- A term may admit multiple types
  - \( \lambda x.x \) can be of type \( \tau \rightarrow \tau \), \( (\tau \rightarrow \tau) \rightarrow (\tau \rightarrow \tau) \), . . .

- Principal type scheme of a term \( M \) — unique type \( s \) such that every other valid type is an “instance” of \( s \)
  - Uniformly replace \( \tau \in s \) by another type
Type checking . . .

- A term may admit multiple types
  - \( \lambda x.x \) can be of type \( \tau \rightarrow \tau \), \((\tau \rightarrow \tau) \rightarrow (\tau \rightarrow \tau)\), . . .

- Principal type scheme of a term \( M \) — unique type \( s \) such that every other valid type is an “instance” of \( s \)
  - Uniformly replace \( \tau \in s \) by another type
  - \( \tau \rightarrow \tau \) is principal type scheme of \( \lambda x.x \)
Type checking . . .

- A term may admit multiple types
  - $\lambda x. x$ can be of type $\tau \rightarrow \tau$, $(\tau \rightarrow \tau) \rightarrow (\tau \rightarrow \tau)$, . . .

- Principal type scheme of a term $M$ — unique type $s$ such that every other valid type is an “instance” of $s$
  - Uniformly replace $\tau \in s$ by another type
  - $\tau \rightarrow \tau$ is principal type scheme of $\lambda x. x$

Theorem

*We can always compute the principal type scheme for any well-typed term in the simply typed $\lambda$-calculus.*
Computability with simple types

- Church numerals are well typed
Computability with simple types

- Church numerals are well typed
- Translations of basic recursive functions (zero, successor, projection) are well-typed
Computability with simple types

- Church numerals are well typed
- Translations of basic recursive functions (zero, successor, projection) are well-typed
- Translation of function composition is well typed
Computability with simple types

- Church numerals are well typed
- Translations of basic recursive functions (zero, successor, projection) are well-typed
- Translation of function composition is well typed
- Translation of primitive recursion is well typed
Computability with simple types

- Church numerals are well typed
- Translations of basic recursive functions (zero, successor, projection) are well-typed
- Translation of function composition is well typed
- Translation of primitive recursion is well typed
- Translation of minimalization requires elimination of recursive definitions
  - Uses untypable expressions of the form $f f$
Computability with simple types

- Church numerals are well typed
- Translations of basic recursive functions (zero, successor, projection) are well-typed
- Translation of function composition is well typed
- Translation of primitive recursion is well typed
- Translation of minimalization requires elimination of recursive definitions
  - Uses untypable expressions of the form $f f$
- Minimalization introduces non terminating computations, but we have strong normalization!
Computability with simple types

- Church numerals are well typed
- Translations of basic recursive functions (zero, successor, projection) are well-typed
- Translation of function composition is well typed
- Translation of primitive recursion is well typed
- Translation of minimalization requires elimination of recursive definitions
  - Uses untypable expressions of the form $f f$
- Minimalization introduces non terminating computations, but we have strong normalization!
- However, there do exist total recursive functions that are not primitive recursive — e.g. Ackermann’s function
Polymorphism

- Simply typed $\lambda$-calculus has explicit types
Polymorphism

- Simply typed $\lambda$-calculus has explicit types
- Languages like Haskell have polymorphic types
  - Compare `id :: a -> a` with $\lambda x.x : \tau \rightarrow \tau$
Polymorphism

- Simply typed $\lambda$-calculus has explicit types

- Languages like Haskell have polymorphic types
  - Compare $\text{id} :: \ a \rightarrow\ a$
    with $\lambda x \cdot x : \tau \rightarrow \tau$

- Second-order polymorphic typed lambda calculus (System F)
  - Jean-Yves Girard
  - John Reynolds
Add type variables, $a$, $b$, \ldots
System F

- Add type variables, \( a, b, \ldots \)
- Use \( i, j, \ldots \) to denote concrete types
Add type variables, \( a, b, \ldots \)

Use \( i, j, \ldots \) to denote concrete types

Type schemes

\[
s ::= a \mid i \mid s \to s \mid \forall a.s
\]
System F

Syntax of second order polymorphic lambda calculus

- Every variable and (type) constant is a term.
Syntax of second order polymorphic lambda calculus

- Every variable and (type) constant is a term.

- If $M$ is a term, $x$ is a variable and $s$ is a type scheme, then $(\lambda x \in s. M)$ is a term.
Syntax of second order polymorphic lambda calculus

- Every variable and (type) constant is a term.
- If $M$ is a term, $x$ is a variable and $s$ is a type scheme, then $(\lambda x \in s.M)$ is a term.
- If $M$ and $N$ are terms, so is $(MN)$.
  - Function application does not enforce type check
System F

Syntax of second order polymorphic lambda calculus

- Every variable and (type) constant is a term.

- If $M$ is a term, $x$ is a variable and $s$ is a type scheme, then $(\lambda x \in s. M)$ is a term.

- If $M$ and $N$ are terms, so is $(MN)$.
  - Function application does not enforce type check

- If $M$ is a term and $a$ is a type variable, then $(\Lambda a. M)$ is a term.
  - Type abstraction
System F

Syntax of second order polymorphic lambda calculus

- Every variable and (type) constant is a term.

- If $M$ is a term, $x$ is a variable and $s$ is a type scheme, then $(\lambda x \in s. M)$ is a term.

- If $M$ and $N$ are terms, so is $(MN)$.
  - Function application does not enforce type check

- If $M$ is a term and $a$ is a type variable, then $(\Lambda a. M)$ is a term.
  - Type abstraction

- If $M$ is a term and $s$ is a type scheme, $(Ms)$ is a term.
  - Type application
Example A polymorphic identity function

\[ \Lambda a. \lambda x \in a.x \]
Example A polymorphic identity function

\[ \Lambda a. \lambda x \in a.x \]

Two $\beta$ rules, for two types of abstraction
Example A polymorphic identity function

\[ \forall a. \lambda x \in a. x \]

Two \( \beta \) rules, for two types of abstraction

\[ (\lambda x \in s. M)N \rightarrow_\beta M \{ x \leftarrow N \} \]
Example A polymorphic identity function

\[ \Lambda a. \lambda x \in a. x \]

Two \(\beta\) rules, for two types of abstraction

1. \((\lambda x \in s. M) N \rightarrow_{\beta} M\{x \leftarrow N\}\)
2. \((\Lambda a. M)s \rightarrow_{\beta} M\{a \leftarrow s\}\)
System F

- System F is also strongly normalizing
System F

- System F is also strongly normalizing

- ...but type inference is undecidable!
  - Given an arbitrary term, can it be assigned a sensible type?
Type inference in System F

- Type of a complex expression can be deduced from types assigned to its parts
Type inference in System F

- Type of a complex expression can be deduced from types assigned to its parts

- To formalize this, define a relation $A \vdash M : s$
  - $A$ is list $\{x_i : t_i\}$ of type “assumptions” for variables
  - Under the assumptions in $A$, the expression $M$ has type $s$. 
Type inference in System F

- Type of a complex expression can be deduced from types assigned to its parts

- To formalize this, define a relation $A \vdash M : s$
  
  - $A$ is list $\{x_i : t_i\}$ of type “assumptions” for variables
  - Under the assumptions in $A$, the expression $M$ has type $s$.

- Inference rules to derive type judgments of the form $A \vdash M : s$
Type inference in System F

Notation
If \(A\) is a list of assumptions, \(A + \{x : s\}\) is the list where

- Assumption for \(x\) in \(A\) (if any) is overridden by the new assumption \(x : s\).
- For any variable \(y \neq x\), assumption does not change
Notation

If $A$ is a list of assumptions, $A + \{x : s\}$ is the list where

- Assumption for $x$ in $A$ (if any) is overridden by the new assumption $x : s$.
- For any variable $y \neq x$, assumption does not change

\[
\frac{A + \{x : s\} \vdash M : t}{A \vdash (\lambda x \in s. M) : s \to t}
\]
Type inference in System F

Notation
If $A$ is a list of assumptions, $A + \{x : s\}$ is the list where

- Assumption for $x$ in $A$ (if any) is overridden by the new assumption $x : s$.
- For any variable $y \neq x$, assumption does not change

\[
\begin{align*}
A + \{x : s\} \vdash M : t \\
\frac{}{A \vdash (\lambda x \in s. M) : s \rightarrow t}
\end{align*}
\]

\[
\begin{align*}
A \vdash M : s \rightarrow t, & \quad A \vdash N : s \\
\frac{}{A \vdash (MN) : t}
\end{align*}
\]
Type inference in System F

Notation
If $A$ is a list of assumptions, $A + \{x : s\}$ is the list where

- Assumption for $x$ in $A$ (if any) is overridden by the new assumption $x : s$.
- For any variable $y \neq x$, assumption does not change

\[
\begin{align*}
A + \{x : s\} &\vdash M : t \\
A &\vdash (\lambda x \in s.M) : s \to t \\
A &\vdash M : s \to t, \ A &\vdash N : s \\
A &\vdash (MN) : t \\
A &\vdash M : s \\
A &\vdash (\forall a.M) : \forall a.s
\end{align*}
\]
Type inference in System F

**Notation**
If $A$ is a list of assumptions, $A + \{x : s\}$ is the list where

- Assumption for $x$ in $A$ (if any) is overridden by the new assumption $x : s$.
- For any variable $y \neq x$, assumption does not change

\[
A + \{x : s\} \vdash M : t \\
\frac{A \vdash (\lambda x \in s.M) : s \rightarrow t}{A \vdash (MN) : t}
\]
\[
\frac{A \vdash M : s \rightarrow t, \quad A \vdash N : s}{A \vdash (MN) : t}
\]
\[
\frac{A \vdash M : s}{A \vdash (\forall a.M) : \forall a.s}
\]
\[
\frac{A \vdash M : \forall a.s}{A \vdash Mt : s\{a \leftarrow t\}}
\]
**Example** Deriving the type of polymorphic identity function

\[ \Lambda a. \lambda x : a. x \]
Example Deriving the type of polymorphic identity function

\( \forall a. \lambda x \in a. x \)

\( x : a \vdash x : a \)
Example Deriving the type of polymorphic identity function

\[ \forall a. \lambda x \in a. x \]

\[ x : a \vdash x : a \]

\[ \vdash (\lambda x \in a. x) : a \rightarrow a \]
Type inference in System F

Example Deriving the type of polymorphic identity function

\[ \forall a. \lambda x \in a. x \]

\[ x : a \vdash x : a \]

\[ \vdash (\lambda x \in a. x) : a \rightarrow a \]

\[ \vdash (\forall a. \lambda x \in a. x) : \forall a. a \rightarrow a \]
Type inference in System F

- Type inference is undecidable for System F
Type inference in System F

- Type inference is undecidable for System F
- ...but we have type-checking algorithms for Haskell, ML, ...!
Type inference in System F

- Type inference is undecidable for System F
- ... but we have type-checking algorithms for Haskell, ML, ...!
- Haskell etc use a restricted version of polymorphic types
  - All types are universally quantified at the top level
Type inference in System F

- Type inference is undecidable for System F
- ...but we have type-checking algorithms for Haskell, ML, ...!
- Haskell etc use a restricted version of polymorphic types
  - All types are universally quantified at the top level
- When we write \texttt{map} :: (a \to b) \to [a] \to [b], we mean that the type is

\[
\texttt{map} :: \forall a, b. (a \to b) \to [a] \to [b]
\]
Type inference in System F

- Type inference is undecidable for System F
- ... but we have type-checking algorithms for Haskell, ML, ...!
- Haskell etc use a restricted version of polymorphic types
  - All types are universally quantified at the top level
- When we write `map :: (a -> b) -> [a] -> [b]`, we mean that the type is
  
  \[
  \text{map} :: \forall a, b. (a \rightarrow b) \rightarrow [a] \rightarrow [b]
  \]

- Also called shallow typing
Type inference in System F

- Type inference is undecidable for System F
- ... but we have type-checking algorithms for Haskell, ML, ...!
- Haskell etc use a restricted version of polymorphic types
  - All types are universally quantified at the top level
- When we write `map :: (a -> b) -> [a] -> [b]`, we mean that the type is
  \[ \text{map} :: \forall a, b. (a \to b) \to [a] \to [b] \]
  - Also called shallow typing
- System F permits deep typing
  \[ \forall a. \left( \forall b. a \to b \right) \to a \to a \]