Lambda Calculus and Type Inference

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
http://www.idt.mdh.se/~blr/

October 13, 2004
The Topics

*Lambda Calculus*: a formal calculus for functions and how to compute with them

*Type Inference*: how to find the possible type(s) of expressions, without explicit typing
Lambda Calculus

Formal calculus

Invented by logicians around 1930

Formal syntax for functions, and function application

Gives a certain “computational” meaning to function application

Theorems about reduction order (which possible subcomputation to execute first)

This is related to call-by-value/call-by-need

Several variations of the calculus
The Simple Untyped Lambda Calculus

A term in this calculus is:

- a variable $x$,
- a lambda-abstraction $\lambda x.e$, or
- an application $e_1 e_2$

Some examples:

$x \quad x \ y \quad x \ x \quad \lambda x.(x \ y) \quad (\lambda x.x) \ y \quad \lambda x.\lambda y.\lambda x.x$

Any term can be applied to any term, no concept of (function) types

Syntax: function application binds strongest, $\lambda x.x \ y = \lambda x.(x \ y) \neq (\lambda x.x) \ y$
Untyped Lambda Calculus with Constants

We can extend the syntax with constants, for instance:

1, 17, +, [], :

We can then write terms closer to Haskell, like

\[ 17 + x \quad \lambda x.(x + y) \quad \lambda l.\lambda x.(l : x) \]

Every Haskell program can be translated into an intermediate form, which essentially is a lambda calculus with constants.

Some constants (like +) can be given a computational meaning, more on this later.
Equivalences

Some lambda-expressions are considered equivalent \((e_1 \equiv e_2)\)

Rule 1: change of name of bound variable gives an equivalent expression \((\text{alpha-conversion})\)

So \(\lambda x. (x \ x) \equiv \lambda y. (y \ y)\)

Quite natural, right?

However, beware of variable capture:

\(\lambda x. \lambda y. x \neq \lambda y. \lambda y. y\)

Renaming must avoid name clashes with locally bound variables

Note that \((\lambda x. \lambda y. x) \ 17 = \lambda y. 17\), whereas \((\lambda y. \lambda y. y) \ 17 = \lambda y. y\). Different!
Beta-reduction

A lambda abstraction applied to an expression can be \textit{beta-reduced}:

\begin{align*}
(\lambda x.x + x) \; 9 & \rightarrow_{\beta} 9 + 9 \\
\end{align*}

Beta-reduction means substitute actual argument for symbolic parameter in function body.

Works also with symbolic arguments:

\begin{align*}
(\lambda x.x + x) \; (\lambda x.y \ z) & \rightarrow_{\beta} (\lambda x.y \ z) + (\lambda x.y \ z) \\
\end{align*}

However, beware of variable capture:

\begin{align*}
(\lambda x.\lambda y. (x + y)) \; y & \nleftrightarrow_{\beta} \lambda y. (y + y) \\
\end{align*}

The fix is to first rename the bound variable \( y \):

\begin{align*}
(\lambda x.\lambda y. (x + y)) \; y & \equiv (\lambda x.\lambda z. (x + z)) \; y \rightarrow_{\beta} \lambda z. (y + z) \\
\end{align*}
Extending the Equivalence

We consider expressions equal if there is a way to convert them into each other, through beta-reductions or “inverse” beta-reductions.

For instance, \((\lambda x.x) \, 17 \equiv (\lambda y.17) \, z\) since

\[(\lambda x.x) \, 17 \rightarrow_\beta 17 \leftarrow_\beta (\lambda y.17) \, z\]
Some Encodings

Many mathematical concepts can be *encoded* in the lambda-calculus. That is, they can be translated into the calculus. For instance, we can encode the *boolean constants*, and a *conditional* (functional if-then-else):

\[
\begin{align*}
\text{TRUE} & = \lambda x.\lambda y. x \\
\text{FALSE} & = \lambda x.\lambda y. y \\
\text{COND} & = \lambda p.\lambda q.\lambda r. (p \ q \ r)
\end{align*}
\]
An example of how \textit{COND} works:

\verb+COND TRUE A B+ \quad \rightarrow_\beta \quad (\lambda p. \lambda q. \lambda r. (p \ q \ r)) \ (\lambda x. \lambda y. x) \ A \ B \\
\quad \rightarrow_\beta \quad (\lambda q. \lambda r. ((\lambda x. \lambda y. x) \ q \ r)) \ A \ B \\
\quad \rightarrow_\beta \quad (\lambda r. ((\lambda x. \lambda y. x) \ A \ r)) \ B \\
\quad \rightarrow_\beta \quad (\lambda x. \lambda y. x) \ A \ B \\
\quad \rightarrow_\beta \quad \lambda y. A \ B \\
\quad \rightarrow_\beta \quad A

Try evaluating \textit{COND FALSE A B} yourself!
Boolean connectives (and, or) can also be encoded

As well as lists, integers, ... 

Actually *anything you can do in a functional language!*
Consider this expression:

\((\lambda x.x \ x) (\lambda x.x \ x)\)

What if we beta-reduce it?

\((\lambda x.x \ x) (\lambda x.x \ x) \rightarrow_\beta (\lambda x.x \ x) (\lambda x.x \ x)\)

Whoa, we got back the same! Scary . . .

Clearly, we can reduce ad infinitum

The lambda-calculus thus contains nonterminating reductions
Recursion

Now consider this expression:
\[ \lambda h. (\lambda x. h (x x)) (\lambda x. h (x x)) \]

Let’s call it \( Y \)

What if we apply it to a function \( f \)?

\[
Y \ f \ = \ \lambda h. (\lambda x. h (x x)) (\lambda x. h (x x)) \ f \\
\rightarrow_{\beta} \ (\lambda x. f (x x)) (\lambda x. f (x x)) \\
\rightarrow_{\beta} \ f \ ((\lambda x. f (x x)) (\lambda x. f (x x))) \\
= \ f \ (Y \ f)
\]

Hmm, we got back \( f \) applied to \( Y \ f \)
\( Y \) is called *fixed-point combinator*

It encodes *recursion*

To see why, consider the recursive definition

\[ x = f \ x \]

The solution is

\[ x = f \ f \ f \ \cdots \]

Likewise,

\[ Y \ f = f \ (Y \ f) = f \ f \ (Y \ f) = f \ f \ f \ \cdots \]

Thus, \( x = Y \ f \)!

Note that *all* recursive definitions can be written on the form \( x = f \ x \)
Reduction Strategies

Any application of a lambda-abstraction in an expression can be beta-reduced

Each such position is called a redex

An expression can contain several redexes

Can you find all redexes in this expression?

$$(\lambda x.((\lambda y. y) \ x) \ ((\lambda y. y) \ x))$$

Try reduce them in different orders!
Does the order of reducing redexes matter?

Well, yes and no:

**Theorem**: if two different reduction orders of the same expression end in expressions that cannot be further reduced, then these expressions must be the same

However, we can have potentially infinite reductions:

\[(\lambda x.y) \ ((\lambda x.x \ x) \ (\lambda x.x \ x))\]

Reducing the “outermost” redex yields \(y\)

But the innermost redex can be reduced infinitely many times – nontermination!

So the order *does* matter, as regards termination anyway!
Normal Order Reduction

Consider this slight adaptation of the previous example:

\[(\lambda x.\lambda x.\,y) \ (\lambda x.\,x) \ ((\lambda x.\,x \ x) \ (\lambda x.\,x \ x))\]

Let us draw it as a tree:

Reducing the “leftmost-outermost” redex twice yields \(y\)
Some other reduction orders do not terminate

This is not a coincidence!

To reduce the leftmost-outermost redex in each step is called *normal order reduction*

**Theorem**: *if there is a reduction order that terminates, then normal order reduction terminates*

Normal order reduction corresponds to call-by-need in functional languages
You have seen that Haskell systems can find types for expressions:

\[ y \ f = f \ (y \ f) \]
\[ y :: a \to a \to a \]

As we have mentioned, the *most general* type is always found
There is an interesting theory behind Haskell-style type inference

To infer means “to prove”, or “to deduce”

A type system is a logic, whose statements are of form “expression $e$ has type $\tau$”

To infer a type means to prove a statement like above

A type inference algorithm finds a type if it exists: it is thus a proof search algorithm

Such an algorithm exists for Haskell’s type system
Hindley-Milner’s Type System

Haskell’s type system extends a simpler type system known as Hindley-Milner’s type system (HM)

HM does not have type classes, and it is defined over a simple lambda-calculus language

Here, we will briefly describe a subset of HM:

Language is lambda-calculus with constants (also known functions, like +)

Constants have typing given in advance

Recursion can be encoded through constant $Y$ (fixed-point combinator)
A language of types:

- type constants (like `Int`)
- type variables $\alpha$
- function types $\tau \rightarrow \tau'$ ($\rightarrow$ is a type constructor)
- type schemes $\forall \alpha. \tau$

Other type constructors (for list types etc) can easily be added

Type schemes like $\forall \alpha. \alpha \rightarrow \alpha$ correspond to polymorphic Haskell types like $\forall \alpha. \alpha \rightarrow \alpha$
Statements

Statements of form $A \vdash e : \tau$, where $e$ is expression and $\tau$ is a type

$A$ is a set of *assumptions*, of form $x : \tau$ (read: “variable $x$ has type $\tau$”)  

$A \vdash e : \tau$ is read “under the assumptions on typings of variables in $A$, the expression $e$ can have type $\tau$”

So in order to prove such a statement, we must “guess” some typings of variables in $e$ and then check that we can give $e$ a consistent type with these assumptions

The key in type inference is to make these “guesses” systematically
Inference Rules

Axioms and allowed proof steps are given as a set of *inference rules*

Each inference rule has a number of *premises* and a *conclusion*

Often written on the form

\[
\frac{\text{premise 1} \ \cdots \ \text{premise } n}{\text{conclusion}}
\]

Example (modus ponens in propositional logic):

\[
\frac{P \quad P \implies Q}{Q}
\]

Axioms are inference rules without premises
Hindley-Milner Inference Rules

A selection of rules from the HM inference system:

\[
A \cup \{x : \tau\} \vdash x : \tau \quad [VAR]
\]

\[
A \cup \{x : \sigma\} \vdash e : \tau \quad \frac{A \vdash \lambda x.e : \sigma \to \tau}{\frac{A \vdash e : \sigma \to \tau}{A \vdash e : \tau}} \quad [ABS]
\]

\[
A \vdash e : \sigma \to \tau \quad A \vdash e' : \sigma \quad \frac{A \vdash e \ e' : \tau}{A \vdash e : \tau} \quad [APP]
\]

\[
A \vdash e : \forall \alpha.\tau \quad \frac{A \vdash e : \tau[\sigma/\alpha]}{A \vdash e : \tau} \quad [SPEC]
\]

(\(\tau[\sigma/\alpha]\) stands for \(\tau\) with \(\sigma\) replacing every occurrence of \(\alpha\))
An Example

Best way to understand inference rules is to see a derivation

Let’s infer a type for \((\lambda y. (\text{tail } y)) \; \text{nil}\)

Extend language of types with list types \([	au]\)

Assume given typings for constants:

\[ A = \{ \text{nil} : \forall \alpha. [\alpha], \text{tail} : \forall \alpha. [\alpha] \rightarrow [\alpha] \} \]
Derivation

\[
T \quad A \vdash \lambda y. (\text{tail } y) : [\gamma] \rightarrow [\gamma] \quad A \vdash \text{nil} : \forall \alpha. [\alpha] \\
\hline
A \vdash (\lambda y. (\text{tail } y)) \text{ nil} : [\gamma]
\]

where

\[
T = \quad A \cup \{ y : [\gamma] \} \vdash \text{tail} : \forall \alpha. [\alpha] \rightarrow [\alpha] \quad A \cup \{ y : [\gamma] \} \vdash y : [\gamma] \\
\hline
A \cup \{ y : [\gamma] \} \vdash \text{tail } y : [\gamma]
\]
Inference Algorithm

There is a classical algorithm for type inference in the HM system

Called \textit{algorithm }\forall

Basically a systematic and efficient way to infer types like we did in the example

The algorithm uses \textit{unification}, remember this when you learn logic programming!

It has been proved that algorithm \forall always yields a \textit{most general type} for any typable expression

“Most general” means that any other possible type for the expression can be obtained from the most general type by instantiating its type variables
A More Practical Type Inference Example

Define

\[
\begin{align*}
\text{length } [] &= 0 \\
\text{length } (x:xs) &= 1 + \text{length } xs
\end{align*}
\]

Derive the most general type for \text{length}!

For simplicity, assume that \(0 : : \text{Int}, 1 : : \text{Int}, \text{and} \ (+: : \text{Int } \rightarrow \text{Int } \rightarrow \text{Int}\)

See next four slides for how to do it …
Type inference can be seen as equation solving: every declaration gives rise to a “type equation” constraining the types for the untyped identifiers.

These equations can be solved to find the types.

In our example, we already know:

\[
\begin{align*}
0 &:: \text{Int} \\
1 &:: \text{Int} \\
(+) &:: \text{Int} \to \text{Int} \to \text{Int} \\
[\ ] &:: [a] \\
(:) &:: b \to [b] \to [b]
\end{align*}
\]

Note different type variable names in types of \([\ ]\) and \((:)\), to make sure they’re not mixed up.
Solving the equation for the first declaration:

\[ \text{length } [] = 0 \]

\[ \text{length} :: c \to d \text{ (since it is applied to one argument)} \]

\[ c = [a] \text{ (from what we know about the type of [])} \]

\[ d = \text{Int} \text{ (since length } [] :: d, 0 :: \text{Int, and both sides of the declaration must have the same type)} \]

Thus, \[ \text{length} :: [a] \to \text{Int} \]

Is this consistent with the second case in declaration of \text{length}?
Second declaration

\[ \text{length (x:xs)} = 1 + \text{length xs} \]

Must first find possible types for \( x, xs, x:xs \)

Assume \( x :: e, xs :: f \)

\( e = b, f = [b] \) (or else \( x:xs \) is not well-typed), then \( x:xs :: [b] \)

Left-hand side: OK if \( [b] = [a] \) (so \( xs :: [a] \) and \( x:xs :: [a] \)), and then \( \text{length (x:xs)} :: \text{Int} \)

Right-hand side: \( xs :: [b], \text{length :: [a] -> Int and xs :: [a]} \) gives \( \text{length xs :: Int} \)

\( 1 :: \text{Int}, \text{length xs :: Int, (+) :: Int -> Int -> Int} \) gives \( 1 + \text{length xs :: Int} \)
Thus type of LHS = type of RHS! We’re done.

Result: \texttt{length :: [a] \to Int}

Must be a most general type since we were careful not to make any stronger assumptions than necessary about any types

(In the formal type system, we would obtain \texttt{\textit{length} : \forall \alpha. [\alpha] \to \text{Int}})
Applications of Type Inference

HM-style type systems used in some advanced functional languages, most notably ML and Haskell

Similar type inference systems can be used for program analysis

Types (not the usual ones) can stand for properties

\[ e : \tau \] then means “expression (program) \( e \) has property \( \tau \)”

Can be used in optimizing compilers

Another application: dimensional analysis, to find whether equations are sound w.r.t. physical dimensions