Type Inference, Higher Order Algebra, and Lambda Calculus

Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University

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

Type Inference, Higher Order Algebra, and Lambda Calculus (revised 2010-12-09)
The Topics

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

*Higher Order Algebra:* a number of laws that the higher order functions like `map`, `fold` etc. obey

*Lambda Calculus:* a formal calculus for functions and how to compute with them
We have seen that the F# compiler can find types for expressions, and declared values:

```fsharp
def length l =
    match l with
    | []  -> 0
    | _::xs -> 1 + length xs

length : 'a list -> int
```

As we have mentioned, the most general type is always found.

How can the compiler do this?
There is an interesting theory behind F#-style type inference

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

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

Often written “$A \vdash e : \tau$”

To infer a type means to *prove* that a statement like above is true

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

Such an algorithm exists for F#’s type system
Logical Systems

A logical system is given by a set of axioms, and inference rules over a language of statements.

A statement is true in the logic if it can be proved in a finite number of steps using these rules.

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

Often written on the form

\[
\text{premise } 1 \quad \cdots \quad \text{premise } n \\
\underline{\text{conclusion}}
\]
Logical Systems

An example of an inference rule (modus ponens in propositional logic):

\[
\frac{P \quad P \implies Q}{Q}
\]
Hindley-Milner’s Type System

F#’s type system extends a simpler type system known as *Hindley-Milner’s type system* (HM)

This system was first invented around 1970

The typing statements have the form $A \vdash e : \tau$, where $A$ is a set of typings for variables, $e$ is an expression, and $\tau$ is a type

**Example:** $\{x : \alpha, f : \alpha \to \beta\} \vdash f \ x : \beta$

The type system of F# is basically the HM type system, with some extensions
Hindley-Milner Inference Rules

A selection of rules from the HM inference system:

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

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

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

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

(You don’t need to learn this: I’m showing it only to let you know what an inference system might look like)
There is a classical algorithm for type inference in the HM system called \textit{algorithm }$\mathcal{W}$.

Basically a systematic and efficient way to infer types.

The algorithm uses \textit{unification}, which is basically a symbolic method to solve equations.

It has been proved that algorithm $\mathcal{W}$ always yields a 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 Type Inference Example

Define

\[
\text{length } l = \\
\text{match } l \text{ with} \\
\text{ | } [] \rightarrow 0 \\
\text{ | } x::xs \rightarrow 1 + \text{length xs}
\]

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

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

These equations can be solved to find the types.

In our example, we already know:

0 : int
1 : int
(+ : 'n -> 'n -> 'n, 'n some numerical type
[] : 'a list
(::) : 'b -> 'b list -> 'b list

Note different type variable names, to make sure they’re not mixed up.
Solving the Equations

Left-hand side:

\[
\text{length } l = \ldots
\]

\[
\text{length : 'c -> 'd (since length is applied to an argument, it has to be a function)}
\]

\[
l : 'c (since length is applied to } l, l \text{ must have the same type as the argument of length)}
\]

\[
\text{length } l : 'd (result of applying length to } l, \text{ so 'd must equal the type of the right-hand side)
\]
Right-hand side, first case for \texttt{l}:

\begin{verbatim}
... 
match \texttt{l} with 
  | [ ]  -> 0 
... 
\end{verbatim}

\texttt{\textquoteleft c = \textquoteleft a list (since \texttt{l} can match [ ], and from the type of [ ])}

Thus, \texttt{length : \textquoteleft a list -> \textquoteleft d}

\texttt{\textquoteleft d = int (since we can have length \texttt{l} = 0, length \texttt{l} : \textquoteleft d, and 0 : int)}

Thus, \texttt{length : \textquoteleft a list -> int}

Is this consistent with the second case in the matching of \texttt{l}?
Right-hand side, second case for $l$:

$$\ldots$$

match $l$ with

$$\ldots$$

$$| \ x::xs \rightarrow 1 + \text{length } xs$$

Must first find possible types for $x$, $xs$, $x::xs$

Assume $x : \ 'e$, $xs : \ 'f$

From the typing of $(::)$ we obtain $'e = 'b$, $'f = 'b$ list, and $x::xs : 'b$ list

$l$ can equal $x::xs$, so OK if $'b$ list $= 'a$ list. Possible only if $'b = 'a$

Then $x : 'a$, $xs : 'a$ list, and $x::xs : 'a$ list
What about $1 + \text{length } xs$?

We have $\text{length : } 'a \text{ list } \rightarrow \text{ int}$, and $\text{xs : } 'a \text{ list}$, which yields $\text{length } xs : \text{ int}$

$l : \text{ int}, \text{length } xs : \text{ int}, (+) : 'n \rightarrow 'n \rightarrow 'n$ gives $'n = \text{ int}$, and then $1 + \text{length } xs : \text{ int}$

Same type as for 0 (first case of match), and $\text{length } l!$ We’re done

Result: $\text{length : } 'a \rightarrow \text{ int}$

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

Find the most general type for int_halve, defined by:

```plaintext
let rec int_halve a l u =
  if u = l+1 || a.[l] = 0.0 || a.[u] = 0.0 then (l,u)
  else let h = (l+u)/2 in
    if a.[h] > 0 then int_halve a l h
    else int_halve a h u
```
Higher Order Algebra

Higher order functions like `map`, `fold`, `>>`, ... obey certain laws

These laws can be compared to laws for aritmetical operators, like

\[ x + (y + x) = (x + y) + z \]

They can be used to transform programs, e.g., optimizing them

They also help understanding the functions better
Some Laws involving List.map

List.map id = id, where id = fun x -> x (the identity function)

List.map (g >> f) = List.map g >> List.map f

List.map f >> List.tl = List.tl >> List.map f

List.map f >> reverse = reverse >> List.map f

List.map f (xs @ ys) = List.map f xs @ List.map f ys
Some Laws involving List.filter

List.filter p >> reverse = reverse >> List.filter p

List.filter p (xs @ ys) = List.filter p xs @ List.filter p ys

map f >> List.filter p = List.filter (f >> p) >> map f
A Property of Fold

If \( \text{op} \) is associative and if \( e \) is left and right unit element for \( \text{op} \), then, for all lists \( xs \):

\[
\text{List.foldBack \( \text{op} \) \( xs \) \( e \) = List.fold \( \text{op} \) \( e \) \( xs \)}
\]
What Can Theorems Like This Be Used For?

A simple example: rewriting to optimize code

reverse >>= filter p >>= map f >>= reverse =
filter p >>= reverse >>= map f >>= reverse =
filter p >>= map f >>= reverse >>= reverse =
filter p >>= map f >>= id =
filter p >>= map f

since obviously

reverse >>= reverse = id
Bird-Meertens Formalism

The identities shown belong to an algebra of list functions

This is known as the Bird-Meertens Formalism

The idea of Bird and Meertens was to do program development by:

• making a specification of the program, using the list primitives, and
• using the identities to transform the specification into an efficient implementation

This attempt has not been overly successful in general, but I think there are niches where the method can be applied

In particular, it has been proposed for programming of parallel computers
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

The calculus consists of a language, and equivalences on expressions in the language. A term in the language is:

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

Some examples:

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

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

Syntax: function application binds strongest, \( \lambda x.x \ y = \lambda x.(x \ y) \neq (\lambda x.x) \ y \)
Lambda Calculus Syntax and Functional Programming

Syntax elements from the lambda calculus have been adopted by higher order functional languages, in particular:

- Function expressions \((\text{fun} \ x \rightarrow \ e)\), from \(\lambda x.e\)
- Function application syntax, and currying: \(f \ e_1 \ e_2\)
We can extend the syntax with constants, for instance:

1, 17, +, [], ::

We can then form terms closer to usual functional languages, like

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

Functional language compilers often first translate into an intermediate form, which essentially is a lambda calculus with constants
Equivalences

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

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

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

Quite natural, right? If we change the name of the formal parameter, the function should still be the same

Example: in F#, \texttt{fun x -> x} and \texttt{fun y -> y} define the same function
Variable Capture

However, beware of variable capture:

\( \lambda x. \lambda y. x \not\equiv \lambda y. \lambda y. y \)

Renaming must avoid name clashes with locally bound variables

Precisely the same problem appears in programming languages:

let f x = let g y = x + y in ...

Here we cannot change \( x \) into \( y \) without precautions. However, OK if we rename \( y \) in \( g \) to \( z \) first:

let f x = let g z = x + z in ...
let f y = let g z = y + z in ...

Same trick is used in lambda calculus: \( \lambda x. \lambda y. x \equiv \lambda x. \lambda z. x \equiv \lambda y. \lambda z. y \)
Beta-reduction

A lambda abstraction applied to an expression can be *beta-reduced*:

\[(\lambda x. x + x) \ 9 \rightarrow_\beta 9 + 9\]

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

A formal model for what happens when *a function is applied to an argument*

Works also with symbolic arguments:

\[(\lambda x.x + x) \ (\lambda y. z) \rightarrow_\beta (\lambda x. y \ z) + (\lambda x. y \ z)\]

Like *inlining* done by optimizing compilers
Variable Capture

However, again beware of variable capture:

$$(\lambda x. \lambda y. (x + y)) \ y \not\rightarrow^\beta \ y \lambda y. (y + y)$$

The fix is to first rename the bound variable $y$:

$$(\lambda x. \lambda y. (x + y)) \ y \equiv (\lambda x. \lambda z. (x + z)) \ y \rightarrow^\beta \ \lambda z. (y + z)$$
The same thing can happen when inlining functions. Example:

\[
\begin{align*}
\text{let } f \ x &= \text{let } g \ y = x + y \text{ in } ... \\
\text{let } h \ y &= f \ (y + 3) \\
\end{align*}
\]

If we want to inline the call to \( f \) in \( g \), then \( g \)'s argument must first be renamed:

\[
\begin{align*}
\text{let } h \ y &= f \ (y + 3) \Rightarrow \\
\text{let } h \ y &= \text{let } g \ z = (y + 3) + z \text{ in } ...
\end{align*}
\]
Some Encodings

Many mathematical concepts can be *encoded* in the (untyped) 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):

\[
\begin{align*}
TRUE &= \lambda x.\lambda y. x \\
FALSE &= \lambda x.\lambda y. y \\
COND &= \lambda p.\lambda q.\lambda r. (p\ q\ r)
\end{align*}
\]

Exercise: make these encodings in F#!
An example of how $COND$ works:

$$
COND \ TRUE \ A \ B \quad \rightarrow_{\beta} \quad (\lambda p. \lambda q. \lambda r. (p \ q \ r)) \ (\lambda x. \lambda y. x) \ A \ B
$$

$$
\rightarrow_{\beta} \quad (\lambda q. \lambda r. ((\lambda x. \lambda y. x) \ q \ r)) \ A \ B
$$

$$
\rightarrow_{\beta} \quad (\lambda r. ((\lambda x. \lambda y. x) \ A \ r)) \ B
$$

$$
\rightarrow_{\beta} \quad (\lambda x. \lambda y. x) \ A \ B
$$

$$
\rightarrow_{\beta} \quad \lambda y. A \ B
$$

$$
\rightarrow_{\beta} \quad A
$$

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

As well as lists, integers, ... Even *recursion* can be encoded as a lambda expression

Actually *anything you can do in a functional language*!

This means that *any functional program* can be translated into the lambda calculus

Thus, lambda calculus serves as a general model for functional languages
Nontermination

Consider this expression:

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

What if we beta-reduce it?

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

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

Clearly, we can reduce ad infinitum

The lambda-calculus thus contains nonterminating reductions
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

There is something called “normal order reduction” in the lambda calculus.

It is a strategy to select which redex to reduce next.

Normal order reduction corresponds to lazy evaluation, or call by need.

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

For functional languages, this means that lazy evaluation always is the “best” in the sense that it terminates whenever the program terminates with some other reduction strategy, like call by value.