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/

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

Type Inference

We have seen that the F# compiler can find types for expressions, and declared values:

```
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 τ”

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

$$\frac{\text{premise } 1 \ \cdots \ \text{premise } n}{\text{conclusion}}$$

Logical Systems

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

$$\frac{P \quad P \Longrightarrow Q}{Q}$$

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 τ 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:

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

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

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

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

(You don't need to learn this: I'm showing it only to let you know what an inference system might look like)

Inference Algorithm

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

Called *algorithm* $\mathcal{W}$

Basically a systematic and efficient way to infer types

The algorithm uses *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

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

Derive the most general type for `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:

```
length l = ...
```

`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` must have the same type as the argument of `length`)

`length l : 'd` (result of applying `length` to `l`. So `'d` must equal the type of the right-hand side)

Right-hand side, first case for `l`:

```
...
match l with
| []    -> 0
....
```

`’c = ’a list` (since `l` can match `[]`, and from the type of `[]`)

Thus, `length : ’a list -> ’d`

`’d = int` (since we can have `length l = 0`, `length l : ’d`, and `0 : int`)

Thus, `length : ’a list -> int`

Is this consistent with the second case in the matching of `l`?

Right-hand side, second case for `l`:

```
...
match l with
....
| x::xs -> 1 + 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 + length xs`?

We have `length : ’a list -> int`, and `xs : ’a list`, which yields `length xs : int`

`1 : int`, `length xs : int`, `(+) : ’n -> ’n -> ’n` gives `’n = int`, and then `1 + length xs : int`

Same type as for `0` (first case of `match`), and `length l`! We’re done

Result: `length : ’a -> 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:

```
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 an be compared to laws for aritmetical operators, like

$$x + (y + z) = (x + y) + z$$

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.tail` = `List.tail >> 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 `op` is associative and if `e` is left and right unit element for `op`, then, for all lists `xs`:

`List.foldBack op xs e` = `List.fold op e xs`

What Can Laws 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
```

How to Prove the Laws

Mathematical laws need mathematical proofs

How can the laws for higher-order functions be proved?

We'll exemplify with the law

`map f (xs @ ys)` = `map f xs @ map f ys`

(Writing `map` for `List.map`)

- First, informal reasoning (to motivate why the law holds)
- Then, a formal proof using induction over lists

An Informal Proof

Let $\mathtt{xs} = [x_1, \ldots, x_m]$, $\mathtt{ys} = [y_1, \ldots, y_n]$

Then

$$\begin{aligned}
\mathtt{map\ f}\ ([x_1, \ldots, x_m] \mathbin{@} [y_1, \ldots, y_n]) &= \mathtt{map\ f}\ ([x_1, \ldots, x_m, y_1, \ldots, y_n]) \\
&= [\mathtt{f}\ x_1, \ldots, \mathtt{f}\ x_m, \mathtt{f}\ y_1, \ldots, \mathtt{f}\ y_n] \\
&= [\mathtt{f}\ x_1, \ldots, \mathtt{f}\ x_m] \mathbin{@} [\mathtt{f}\ y_1, \ldots, \mathtt{f}\ y_n] \\
&= \mathtt{map\ f}\ [x_1, \ldots, x_m] \mathbin{@} \\
&\quad\ \mathtt{map\ f}\ [y_1, \ldots, y_n]
\end{aligned}$$

That is,

$$\mathtt{map\ f}\ (\mathtt{xs} \mathbin{@} \mathtt{ys}) = \mathtt{map\ f\ xs} \mathbin{@} \mathtt{map\ f\ ys}$$

Q.E.D.

An Formal Proof

If you really want to be sure . . .

A proof by **induction**

The proof will be over the **structure of lists**

It will use the **recursive definitions** of `@` and `map`

Equality of Lists

The law states that two lists are equal

But when are two lists equal?

This is the definition:

$$\begin{array}{rcl} [\,] & = & [\,] \\ [\,] & \neq & \texttt{x::xs} \\ \texttt{x::xs} & \neq & [\,] \\ \texttt{x::xs} & = & \texttt{y::ys} \iff \texttt{x} = \texttt{y} \wedge \texttt{xs} = \texttt{ys} \end{array}$$

This is a *mathematical* definition

It is *recursive*. Can be directly implemented by a recursive function

Proof by Induction

Have you ever performed proofs by induction? (You should have. . .)

They prove properties that hold for *all non-negative integers*

For instance, $\forall n. \sum_{i=0}^{n} i = n(n+1)/2$

Exercise: prove this property by induction!

But first, let's check out next slide . . .

The Induction Principle for Natural Numbers

Goal: show that the property P is true for all natural numbers (whole numbers ≥ 0)

Proof by induction goes like this:

1. Show that P holds for 0 (the *base case*)
2. Show, for all natural numbers n, that if P holds for n then P holds also for $n+1$ (the *induction step*)
3. Conclude that P holds for all n

To prove 2 one typically assumes that $P(n)$ is true (the *induction hypothesis*), then shows that $P(n+1)$ follows

Why does Induction over the Natural Numbers Work?

The set of natural numbers $\mathbf{N}$ is an *inductively defined set*

$\mathbf{N}$ is defined as follows:

- $0 \in \mathbf{N}$
- $\forall x. x \in \mathbf{N} \implies s(x) \in \mathbf{N}$ (the *successor* of x, i.e., $x+1$)

$$\begin{array}{ccccccccc} 0 & \rightarrow & s(0) & \rightarrow & s(s(0)) & \rightarrow & s(s(s(0))) & \rightarrow & \cdots \\ 0 & & 1 & & 2 & & 3 & & \cdots \end{array}$$

Proofs by induction follow the structure of the inductively defined set!

The Inductively Defined Set of Lists

Inductively defined sets are typically sets of *infinitely* many *finite* objects

The set `'a list` of (finite) lists with elements of type `'a`:

1. `[]` $\in$ `'a list`
2. `x` $\in$ `'a` $\land$ `xs` $\in$ `'a list` $\Longrightarrow$ `x::xs` $\in$ `'a list`

Note similarity with the set of natural numbers!

Also cf. the following type declaration (in “pseudo”-F#):

```
type 'a list = [] | (::) of 'a * 'a list
```

An Induction Principle for Lists

Proof by induction for (finite) lists goes like this:

1. Show that P holds for `[]`
2. Show, for all finite lists `xs` $\in$ `'a list` and all possible list elements `x` $\in$ `'a`, that if P holds for `xs` then P holds also for `x::xs`
3. Conclude that P holds for all finite lists in `'a list`

The Formal Proof

Now let’s formally prove our equality

Prove that:

$$\forall \texttt{xs}.\forall \texttt{ys}.\forall \texttt{f}.[\texttt{map f (xs @ ys)} = \texttt{map f xs @ map f ys}]$$

What induction hypothesis to use? This is often the tricky question!

General rule: look at the function definitions, and try to formulate the induction hypothesis so it matches the recursive structure!

Function Definitions

We recall:

```
[] @ ys        = ys
(x :: xs) @ ys = x :: (xs @ ys)

map f []      = []
map f (x::xs) = f x :: map f xs
```

(“Mathematical” case-by-case versions of the function definitions)

@ recurses over its first argument (`xs` in the statement to prove)

Thus, let’s do the induction over `xs`

Induction Hypothesis

This is then our induction hypothesis:

$$P(\texttt{xs}) = \forall \texttt{ys}.\forall \texttt{f}.[\texttt{map f } (\texttt{xs} \mathbin{@} \texttt{ys}) = \texttt{map f xs} \mathbin{@} \texttt{map f ys}]$$

If we can prove $\forall \texttt{xs}.P(\texttt{xs})$, then we have proved that the law holds!

We will now prove the following:

1. $P([\,])$ (base case)
2. $\forall \texttt{x}.\forall \texttt{xs}.[P(\texttt{xs}) \implies P(\texttt{x::xs})]$ (induction step)

By the induction principle for lists, this will prove $\forall \texttt{xs}.P(\texttt{xs})$

Base Case

$$P([\,]) = \forall \texttt{ys}.\forall \texttt{f}.[\texttt{map f } ([\,] \mathbin{@} \texttt{ys}) = \texttt{map f } [\,] \mathbin{@} \texttt{map f ys}]$$

Assume any `ys`, `f`

Let's show that the LHS equals the RHS:

$$\begin{array}{rcl} \text{LHS} & = & \texttt{map f } ([\,] \mathbin{@} \texttt{ys}) \\ & = & \texttt{map f ys} \\ \text{RHS} & = & \texttt{map f } [\,] \mathbin{@} \texttt{map f ys} \\ & = & [\,] \mathbin{@} \texttt{map f ys} \\ & = & \texttt{map f ys} \end{array}$$

Thus LHS $=$ RHS, and $P([\,])$ holds

Induction step

We want to prove

$$P(\texttt{x::xs}) = \forall \texttt{ys}.\forall \texttt{f}.[\texttt{map f } ((\texttt{x} :: \texttt{xs}) \mathbin{@} \texttt{ys}) = \texttt{map f } (\texttt{x} :: \texttt{xs}) \mathbin{@} \texttt{map f ys}]$$

We are allowed to use $P(\texttt{xs})$ in the proof. Assume any `ys`, `f`. Then,

$$\begin{array}{rcl} \text{LHS} & = & \texttt{map f } ((\texttt{x} :: \texttt{xs}) \mathbin{@} \texttt{ys}) \\ & = & \texttt{map f } (\texttt{x} :: (\texttt{xs} \mathbin{@} \texttt{ys})) \\ & = & \texttt{f x} :: \texttt{map f } (\texttt{xs} \mathbin{@} \texttt{ys})) \\ & = & \text{(induction hypothesis)} \\ & = & \texttt{f x} :: (\texttt{map f xs} \mathbin{@} \texttt{map f ys}) \\ & = & (\texttt{f x} :: \texttt{map f xs}) \mathbin{@} \texttt{map f ys} \\ & = & \texttt{map f } (\texttt{x} :: \texttt{xs}) \mathbin{@} \texttt{map f ys} \\ & = & \text{RHS} \end{array}$$

Conclusion

We showed the base case $P([\,])$, and the induction step $P(\texttt{xs}) \implies P(\texttt{x::xs})$

We can thus conclude that $\forall \texttt{xs}.P(\texttt{xs})$

That is, the law holds

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 (Curry, Schönfinkel, and others)

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

H. B. Curry

M. Schönfinkel

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:

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

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 (`fun x -> e`), from $\lambda x.e$
- Function application syntax, and currying: `f e1 e2`

Untyped Lambda Calculus with Constants

We can extend the syntax with constants, for instance:

1, 17, $+$, $[\,]$, $::$

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

$17 + x \qquad \lambda x.(x + y) \qquad \lambda l.\lambda x.(l :: x)$

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 (*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#, `fun x -> x` and `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 e 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:

$(\lambda x.x + x)\ (\lambda x.y\ z) \rightarrow_\beta (\lambda x.y\ z) + (\lambda x.y\ z)$

Like *inlining* done by optimizing compilers

Variable Capture

However, again beware of variable capture:

$(\lambda x.\lambda y.(x+y))\ y \not\to_\beta \lambda y.(y+y)$

The fix is to first rename the bound variable y:

$(\lambda x.\lambda y.(x+y))\ y \equiv (\lambda x.\lambda z.(x+z))\ y \to_\beta \lambda z.(y+z)$

The same thing can happen when inlining functions. Example:

```
let f x = let g y = x + y in ...
let h y = f (y + 3)
```

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

```
let h y = f (y + 3) =>
let h y = let g z = (y + 3) + z in ...
```

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):

$$\begin{array}{rcl} TRUE & = & \lambda x.\lambda y.x \\ FALSE & = & \lambda x.\lambda y.y \\ COND & = & \lambda p.\lambda q.\lambda r.(p\ q\ r) \end{array}$$

Exercise: make these encodings in F#!

An example of how $COND$ works:

$$\begin{array}{rcl} COND\ TRUE\ A\ B & \to_\beta & (\lambda p.\lambda q.\lambda r.(p\ q\ r))\ (\lambda x.\lambda y.x)\ A\ B \\ & \to_\beta & (\lambda q.\lambda r.((\lambda x.\lambda y.x)\ q\ r))\ A\ B \\ & \to_\beta & (\lambda r.((\lambda x.\lambda y.x)\ A\ r))\ B \\ & \to_\beta & (\lambda x.\lambda y.x)\ A\ B \\ & \to_\beta & \lambda y.A\ B \\ & \to_\beta & A \end{array}$$

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:

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

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

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