Advances in Programming Languages

Lecture 6: Higher Types

Ian Stark

School of Informatics
The University of Edinburgh

Thursday 4 October 2018
Semester 1 Week 3

THE UNIVERSITY *of* EDINBURGH

Topic: Some Types in Programming Languages

The current block of lectures look at some uses of *types*.

- Terms and Types
- Parameterized Types and Polymorphism
- Higher Polymorphism
- Higher Types
- Dependent Types

The study of *Type Theory* is part of logic and the foundations of mathematics. However, many aspects of it apply directly to programming languages, and research in type systems has for many decades been an active route for the exchange of new ideas between computer science and mathematics.

Outline

1. Opening
2. Hindley-Milner and more
3. System F
4. Datatypes
5. Beyond System F
6. Closing

Outline

1. Opening
2. Hindley-Milner and more
3. System F
4. Datatypes
5. Beyond System F
6. Closing

Homework

Really Do This

Before the next lecture find an online tutorial for each of the assignment topics. Send me your list of five links and I will summarize them for the class.

Use them to help choose your topic.

Watch This

https://is.gd/weirich_types (Video, 29m33s)

Dependent Typing, Extending Haskell, Type System Research: Interview by InfoQ

Stephanie Weirich
Professor of Computer and Information Science, University of Pennsylvania

Review

Parameterized types like Queue<String> express families of types with common structure, applying a type constructor to one or more type parameters.

Behavioural subtyping is based on Liskov's principle of substitutivity that S is a subtype of T if and only if any S can be used in place of a T.

Variance describes subtyping for parameterized types, where type parameters may be covariant, contravariant or invariant.

Hindley-Milner is a way to type parametric polymorphism in the lambda-calculus, introducing type schemes to generalize types and let-binding syntax to use polymorphic functions.

Type Inference makes it possible to write strongly-typed polymorphic code that is expressive but uncluttered by type annotations; while "Algorithm W" automatically identifies a principle type that is the most general type possible for a term.

Rank-2 types and beyond describe things not reachable in Hindley-Milner.

Outline

1 Opening

2 Hindley-Milner and more

3 System F

4 Datatypes

5 Beyond System F

6 Closing

Types for Parametric Polymorphic Code

OCaml

```
reverse : 'a list -> 'a list

length : 'a list -> int

# let rec map f = function
           | [ ] -> [ ]
           | (x :: xs) -> (f x) :: (map f xs) ;;

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

When compiled, the functions reverse, length and map will each use exactly the same code for any argument type.

Types for Parametric Polymorphic Code

Java

```
static void rotate(List<?> list, int distance)       // In java.util.Collections

static void shuffle(List<?> list)                     // Uses a default randomness source

static <E> List<E> heapSort(List<E> elements) {
    Queue<E> queue = new PriorityQueue<E>(elements);
    List<E> result = new ArrayList<E>();

    while (!queue.isEmpty()) result.add(queue.remove());

    return result;
} // Code from https://docs.oracle.com/javase/tutorial/collections/interfaces/queue.html
```

When compiled, the methods rotate, shuffle and heapSort will use exactly same code for any argument type.

Polymorphic Types in the Lambda Calculus

$$\mathsf{fst} : \forall\alpha,\beta.(\alpha\times\beta)\to\alpha$$

$$\begin{aligned}
\mathsf{swap} &= \lambda p.(\mathsf{snd}\,p, \mathsf{fst}\,p) && : \forall\alpha,\beta.(\alpha\times\beta)\to(\beta\times\alpha)\\
\mathsf{identity} &= \lambda x.x && : \forall\alpha.\alpha\to\alpha\\
\mathsf{apply} &= \lambda f.(\lambda x.(f x)) && : (\alpha\to\beta)\to\alpha\to\beta\\
\mathsf{compose} &= \lambda f.\lambda g.\lambda x.g(f\,x) && : \forall\alpha,\beta,\gamma.(\alpha\to\beta)\to(\beta\to\gamma)\to(\alpha\to\gamma)
\end{aligned}$$

Generalise types τ to *type schemes* σ which *quantify* over *type variables* α, β, γ,...

$$\sigma ::= \tau \mid \forall\alpha.\sigma$$

Type schemes cannot have the for-all quantifier inside types, just at the outer level.

Concrete types τ are *instances* of a type scheme σ.

Also sometimes referred to as *polytype* σ and *monotype* τ.

Checking Polymorphic Types

Rules for Polymorphic Types

Generalize
$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall\alpha.\sigma} \quad \alpha \notin \mathsf{free}(\Gamma)$$

Specialize
$$\frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M : \sigma\{\tau/\alpha\}} \quad \mathsf{free}(\sigma) \cap \mathsf{free}(\tau) = \emptyset$$

Let-binding
$$\frac{\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash M_2 : \sigma_2}{\Gamma \vdash \mathsf{let}\, x = M_1 \,\mathsf{in}\, M_2 : \sigma_2}$$

Notice that we cannot form a lambda-abstraction with a polymorphically-typed variable, but instead have to use a new *let-binding* syntax.

$$\mathsf{let}\, x = M \,\mathsf{in}\, N$$

This relates to the way type schemes only allow the for-all quantifier $\forall$ at the outer level.

Inferring Polymorphic Types

We cannot abstract variables of polymorphic type into a lambda term, but instead have to use a *let-binding* syntax that indicates where a term is to be used polymorphically:

$$\mathsf{let}\, x = M \,\mathsf{in}\, N$$

This restriction makes it possible to perform *type inference*: given a lambda term with no types, it is possible to work out a type scheme that is:

- Correct — it can be checked using the rules; and
- The *most general* or *principal* type scheme — all other correct types are instances of it.

This is known as the *Hindley-Milner* type system, and the original method for type inference is called "Algorithm W".

Hindley-Milner forms the basis of types in Haskell and all ML-family languages languages like OCaml and F#. They offer an excellent trade-off between expressivity (lots of terms have useful types) and practicality (type inference is always possible, and the algorithm is efficient).

Comparing Sort Algorithms

Suppose we have a collection of functions, all implementing different sorting algorithms.

$$\mathsf{Sorter} \;=\; \forall\alpha.(\alpha \to \alpha \to \mathsf{bool}) \to \mathsf{list}\,\alpha \to \mathsf{list}\,\alpha$$

$$\mathsf{bubbleSorter}, \mathsf{quickSorter}, \mathsf{heapSorter}, \mathsf{mergeSorter}, \mathsf{bogoSorter}, \ldots \quad : \quad \mathsf{Sorter}$$

The Sorter type scheme captures how each algorithm can be applied to different types of list.

Here's a function that takes a Sorter and tries it out on a few cases.

$$\begin{aligned}
\mathsf{simpleTester} \;=\;& \\
&\lambda\,\mathsf{sorter}\;.\;((\mathsf{sorter}\;\mathsf{greaterThan}\;[5, 22, 2]) == [2, 5, 22]) \\
&\qquad\qquad \mathsf{and}((\mathsf{sorter}\;\mathsf{lessThan}\;[5, 22, 2]) == [22, 5, 2] \\
&\qquad\qquad \mathsf{and}((\mathsf{sorter}\;\mathsf{dictionaryBefore}\;[\text{"sort"}, \text{"test"}]) == [\text{"sort"}, \text{"test"}]
\end{aligned}$$

The simpleTester takes a single polymorphic argument, the sorter, and uses it at multiple types. This is *rank-2 polymorphism*.

Comparing Sort Algorithms

In some cases it is possible to automatically infer rank-2 polymorphic types.

$$\mathsf{Tester} \;=\; (\mathsf{Sorter} \to \mathsf{bool}) \;=\; (\forall\alpha.(\alpha \to \alpha \to \mathsf{bool}) \to \mathsf{list}\,\alpha \to \mathsf{list}\,\alpha) \to \mathsf{bool}$$

$$\mathsf{simpleTester} \;:\; \mathsf{Tester}$$

What if we go higher? Suppose we want to build the sorter comparison game and apply a whole range of tests to different sorters?

$$\mathsf{testManySorters} = \lambda\,\mathsf{sorters}\,.\,\lambda\,\mathsf{testers}\,.\,(\mathsf{tabulate\ testers\ sorters})$$

$$\mathsf{testManySorters}\ [\mathsf{bubbleSorter},\ \mathsf{quickSorter},\ \mathsf{heapSorter}]\ [\mathsf{yourTester},\ \mathsf{myTester}]$$

What type does this have? We are now beyond even rank-2 polymorphism, and cannot manage without significantly more explicit type annotations.

The Sorter Comparison Game

$$\mathsf{testManySorters} = \lambda\,\mathsf{sorters}\,.\,\lambda\,\mathsf{testers}\,.\,(\mathsf{tabulate\ testers\ sorters})$$

$$\mathsf{testManySorters}\ [\mathsf{bubbleSorter},\ \mathsf{quickSorter},\ \mathsf{heapSorter}]$$
$$[\mathsf{yourSorterTester},\ \mathsf{mySorterTester}]$$

What type does this have? We are now beyond even rank-2 polymorphism, and cannot manage without significantly more explicit type annotations.

$$\begin{aligned}
\mathsf{testManySorters}\ :\ & \mathsf{list}\,(\mathsf{Sorter}) \to \mathsf{list}\,(\mathsf{Tester}) \to \mathsf{list}\,(\mathsf{list}\,\mathsf{bool}) \\
= {} & \mathsf{list}(\forall\alpha.(\alpha \to \alpha \to \mathsf{bool}) \to \mathsf{list}\,\alpha \to \mathsf{list}\,\alpha) \\
& \to \mathsf{list}((\forall\alpha.(\alpha \to \alpha \to \mathsf{bool}) \to \mathsf{list}\,\alpha \to \mathsf{list}\,\alpha) \to \mathsf{bool}) \\
& \to \mathsf{list}\,(\mathsf{list}\,\mathsf{bool})
\end{aligned}$$

Outline

1 Opening

2 Hindley-Milner and more

3 System F

4 Datatypes

5 Beyond System F

6 Closing

System F

The *polymorphic lambda-calculus*, also known as the *second-order lambda-calculus*, or *System F*, was discovered independently by the logician Jean-Yves Girard and the computer scientist John Reynolds.

In System F a polymorphic term is a function with a type as a parameter. For example:

$$\text{identity} \;=\; \Lambda X.(\lambda x{:}X\,.\,x) \qquad : \forall X.(X \to X)$$

With this definition:

$$\text{identity}\;A\;M \xrightarrow{\beta} M \qquad \text{for any } M : A.$$

Moreover, because $\forall X.(X \to X)$ is a System F type, we even have:

$$\text{identity}\;(\forall X.(X \to X))\;\text{identity} \xrightarrow{\beta} \text{identity}\,.$$

The fact that $\forall X$ ranges over all possible types, even the type being defined at the time, is known as *impredicativity*.

Hindley-Milner is *predicative*

Change of Notation Metavariables

When describing Hindley-Milner types, the type schemes and type variables were written with Greek letters (τ, σ, α). To distinguish System F this lecture moves to Roman letters for type (meta)variables.

Notation

Terms		Types	
Variables	x, y, z	Variables	X, Y, Z
Terms	$M, N, \ldots$	Types	$A, B, C, \ldots$
Term definitions, constants, constructors	pair, fst, snd, uncapitalisedwords	Type definitions, constants, constructors	Product, Sum, CapitalisedWords

All types and terms will be written with Church-style explicit types as in $(\lambda x{:}A.M)$.

Declarations use a *type variable context* $\Delta = \{X_1, X_2, \ldots\}$ and *term variable context* $\Gamma = \{x_1 : A_1, x_2 : A_2, \ldots\}$.

Rules for Types and Terms in System F

Types

Type Variable

$$\frac{}{\Delta \vdash \mathsf{Type}\ X} \quad X \in \Delta$$

Function Type

$$\frac{\Delta \vdash \mathsf{Type}\ A \qquad \Delta \vdash \mathsf{Type}\ B}{\Delta \vdash \mathsf{Type}\ A \to B}$$

For-All Type

$$\frac{\Delta, X \vdash \mathsf{Type}\ A}{\Delta \vdash \mathsf{Type}\ \forall X.A}$$

Terms

Variable

$$\frac{}{\Delta; \Gamma \vdash x : A} \quad x : A \in \Gamma$$

Abstraction

$$\frac{\Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash (\lambda x{:}A.M) : A \to B}$$

Application

$$\frac{\Delta; \Gamma \vdash F : A \to B \qquad \Delta; \Gamma \vdash M : A}{\Delta; \Gamma \vdash FM : B}$$

Type Abstraction

$$\frac{\Delta, X; \Gamma \vdash M : A}{\Delta; \Gamma \vdash \Lambda X.M : \forall X.A}$$

Type Application

$$\frac{\Delta \vdash \mathsf{Type}\ A \qquad \Delta; \Gamma \vdash M : \forall X.B}{\Delta; \Gamma \vdash MA : B\{A/X\}}$$

Rules for Reduction of Terms in System F

The basic lambda-calculus rewrite rule of *beta-reduction*, where a function is applied to an argument, in System F now has two cases.

Beta-Reduction

Type Application

$$(\Lambda X.M)\,A \;\longrightarrow\; M\{A/X\}$$

Term Application

$$(\lambda x{:}A.M)\,N \;\longrightarrow\; M\{N/x\}$$

We write

$$M \overset{\beta}{\Longrightarrow} N$$

to indicate that term M reduces to N in zero or more steps of beta-reduction.

Some System F Types

System F provides enough machinery to typecheck all those sorters and testers; although this also means passing around types explicitly.

$$\mathsf{Sorter} \;=\; \forall X.(X \to X \to \mathsf{Bool}) \to \mathsf{List}\,X \to \mathsf{List}\,X$$

$$\mathsf{bubbleSorter}, \mathsf{quickSorter}, \mathsf{heapSorter}, \mathsf{mergeSorter}, \mathsf{bogoSorter}, \ldots \quad : \quad \mathsf{Sorter}$$

$$\begin{aligned}
&\mathsf{simpleTester} \;=\\
&\lambda\,\mathsf{sorter}{:}\mathsf{Sorter}\,.\;((\mathsf{sorter}\ \mathsf{num}\ \mathsf{greaterThan}\ [5,22,2]) == [2,5,22])\\
&\qquad\qquad \mathsf{and}((\mathsf{sorter}\ \mathsf{num}\ \mathsf{lessThan}\ [5,22,2]) == [22,5,2]\\
&\qquad\qquad \mathsf{and}((\mathsf{sorter}\ \mathsf{string}\ \mathsf{dictionaryBefore}\ [\text{"sort"},\text{"test"}]) == [\text{"sort"},\text{"test"}]
\end{aligned}$$

$$\begin{aligned}
\mathsf{simpleTester} \;:\; \mathsf{Tester} &= \mathsf{Sorter} \to \mathsf{Bool}\\
&= (\forall X.(X \to X \to \mathsf{Bool}) \to \mathsf{List}\,X \to \mathsf{List}\,X) \to \mathsf{Bool}
\end{aligned}$$

Some System F Types

Now that sorters and testers are explicitly polymorphic, the high-level operation of tabulating results can be done without any need to see these low-level details.

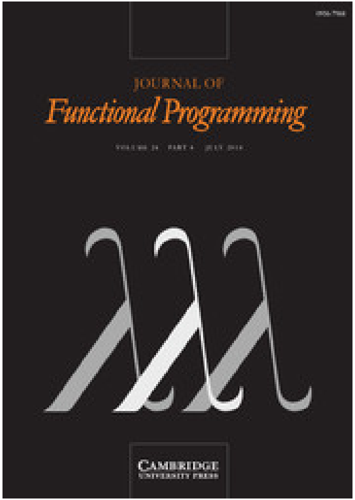
$$\mathsf{testManySorters} = \lambda\, \mathsf{sorters} : \mathsf{List}\,(\mathsf{Sorter})\,.\,\lambda\, \mathsf{testers} : \mathsf{List}\,(\mathsf{Tester})\,.\,(\mathsf{tabulate\ testers\ sorters})$$

$$\mathsf{testManySorters}\,[\,\mathsf{bubbleSorter},\ \mathsf{quickSorter},\ \mathsf{heapSorter}\,]\,[\,\mathsf{yourTester},\ \mathsf{myTester}\,]$$

$$\mathsf{testManySorters} : \mathsf{List}\,(\mathsf{Sorter}) \to \mathsf{List}\,(\mathsf{Tester}) \to \mathsf{List}\,(\mathsf{List\,bool})$$

If You Liked That…

Chris Okasaki
Even higher-order functions for parsing or *Why would anyone ever want to use a sixth-order function?*
Journal of Functional Programming 8(2):195–199, March 1998
DOI: 10.1017/S0956796898003001



Outline

1. Opening
2. Hindley-Milner and more
3. System F
4. Datatypes
5. Beyond System F
6. Closing

Some Datatypes

Basic Datatype Constructors

Function Space	$\lambda x{:}A.M : A \to B$	As in functions, lambdas, procedures, methods,...
Product	$(M, N) : A \times B$	As in product, record, struct,...
Sum	inl, inr : $A + B$	As in sum, variant, union,...

Of these, the simplest version of System F includes only function spaces. The others can be added, but — perhaps surprisingly — they can also be defined within System F already by using function spaces and polymorphic type abstraction.

Encoding Products in System F

Product Type and Pairing

$$\mathsf{Prod}\ X\ Y = \forall Z.((X \to Y \to Z) \to Z)$$

$$\mathsf{pair} = \Lambda X.\Lambda Y.(\lambda x{:}X.\lambda y{:}Y.(\Lambda Z.\lambda f{:}(X \to Y \to Z).(f\ x\ y)))$$

$$\mathsf{pair} : \forall X.\forall Y.(X \to Y \to \mathsf{Prod}\ X\ Y)$$

$$\mathsf{pair}\ A\ B\ M\ N \xrightarrow{\beta} \Lambda Z.\lambda f{:}(A \to B \to Z).f\ M\ N$$

First Projection

$$\mathsf{fst} = \Lambda X.\Lambda Y.\lambda p{:}(\mathsf{Prod}\ X\ Y).p\ X\ (\lambda x{:}X.\lambda y{:}Y.x)$$

$$\mathsf{fst} : \forall X.\forall Y.\mathsf{Prod}\ X\ Y \to X$$

$$\mathsf{fst}\ A\ B\ (\mathsf{pair}\ A\ B\ M\ N) \xrightarrow{\beta} M$$

Syntactic Sugar

$$A \times B = \mathsf{Prod}\ A\ B$$

$$(M, N)_{A,B} = \mathsf{pair}\ A\ B\ M\ N : A \times B$$

$$\mathsf{fst}_{A,B} = \mathsf{fst}\ A\ B : A \times B \to A$$

Exercises for the Reader

Based on the preceding encoding for products in System F:

- Write out a definition for second projection "snd";
- Show that it has the right type, and reduces with

$$\mathsf{snd}\ A\ B\ (\mathsf{pair}\ A\ B\ M\ N) \xrightarrow{\beta} N$$

- Define terms inl and inr and case for the following definition of sum types:

$$\mathsf{Sum}\ X\ Y = \forall Z.((X \to Z) \to (Y \to Z) \to Z)$$

- What is the type corresponding to $(\forall X.X \to X)$? What about $(\forall X.X)$?

Outline

1. Opening
2. Hindley-Milner and more
3. System F
4. Datatypes
5. Beyond System F
6. Closing

Beyond System F

System F is a powerful and expressive type system, but it is just the start of a whole panoply of type features.

System $F_{<:}$ (F-sub) introduces *bounded quantification* $\forall X{<}A.B$.

> Java uses this in declarations like **class** A<T **extends** String> ...

The more elaborate *F-bounded quantification* is $\forall X{<}F(X).B$ for any type constructor $F(-)$.

> Java uses this too, in **class** A<T **extends** Comparable<T>> ...

System F_2 introduces lambda-abstraction for types, not just terms; for example:

$$(\lambda(X{:}*).\lambda(Y{:}*).(X \times Y \times Y)) : * \to * \to * \,.$$

With *System* F_ω we get abstraction over type operators of higher kinds; for example:

$$(\lambda(F{:}(* \to * \to *)).\lambda(X{:}*).(F\,X\,X)) : (* \to * \to *) \to * \to * \,.$$

And the *existential type* $\exists X.A$ is dual to the universal $\forall X.A$, but can be encoded using it. . .

Outline

1. Opening
2. Hindley-Milner and more
3. System F
4. Datatypes
5. Beyond System F
6. Closing

Homework

1. Do This

Work through those "Exercises for the Reader" on encoding products and sums in System F.

2. Write This

The outline draft for your written coursework assignment. Aim to have by Monday's lecture *either* your "Hello, World!" screenshot in the Example section *or* a paragraph for each of three references in the Resources section.

Extensions

Pick a strongly-typed programming language then try writing (and typechecking) two sorters, two testers, and code to testManySorters.

Find out how { Java, Scala, C#, Haskell, ... } handles type { variance, bounds, quantification, kinds }.