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

- Terms and Types
- Parameterized Types and Polymorphism
- **Higher Polymorphism**
- 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 Subtyping and Polymorphism
3 Beyond Hindley-Milner
4 System F
5 Datatypes
6 Beyond System F
7 Closing
Outline

1 Opening

2 Subtyping and Polymorphism

3 Beyond Hindley-Milner

4 System F

5 Datatypes

6 Beyond System F

7 Closing
Parameterized types let us express families of types with common structure, building a complex structured type by applying a *type constructor* to one or more *type parameters*.

### Examples of Parameterized Types

<table>
<thead>
<tr>
<th>Language</th>
<th>Type in Type</th>
<th>Constructor</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Java</strong></td>
<td>Set&lt;String&gt;</td>
<td>Set</td>
<td>String</td>
</tr>
<tr>
<td><strong>Haskell</strong></td>
<td>Tree int</td>
<td>Tree</td>
<td>int</td>
</tr>
<tr>
<td><strong>OCaml</strong></td>
<td>(bool $\to$ bool) list</td>
<td>list</td>
<td>bool $\to$ bool</td>
</tr>
</tbody>
</table>
Polymorphism enables code to act on values of many types. For *ad-hoc polymorphism*, method dispatch gives different actions on different types. With *parametric polymorphism*, a single piece of code acts in the same way on many different argument types.

### OCaml

```ocaml
let rec map f = function ...
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

### Java generics

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

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

static <E> List<E> heapSort(List<E> elements) { ... }
```

https://docs.oracle.com/javase/tutorial/collections/interfaces/queue.html
Hindley-Milner types, type schemes $\sigma$ and type inference make it possible to write strongly-typed polymorphic code that is expressive but uncluttered by type annotations: while Algorithm $W$ automatically identifies a principal type, the most general type possible for a term.

\[
\text{swap} = \lambda p. (\text{snd } p, \text{fst } p) : \forall \alpha, \beta. (\alpha \times \beta) \rightarrow (\beta \times \alpha)
\]

\[
\text{identity} = \lambda x. x : \forall \alpha. \alpha \rightarrow \alpha
\]

\[
\text{apply} = \lambda f. (\lambda x. (fx)) : (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta
\]

\[
\text{compose} = \lambda f. \lambda g. \lambda x. g(f x) : \forall \alpha, \beta, \gamma. (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \gamma)
\]

let
\[
\text{compose} = \lambda f. \lambda g. \lambda x. g(f x)
\]
\[
\text{swap} = \lambda p. (\text{snd } p, \text{fst } p)
\]
in
\[
\text{compose swap swap} : \forall \alpha, \beta. (\alpha \times \beta) \rightarrow (\alpha \times \beta)
\]
Outline

1. Opening
2. Subtyping and Polymorphism
3. Beyond Hindley-Milner
4. System F
5. Datatypes
6. Beyond System F
7. Closing
Homework (1/2)

Read This

Philip Wadler
Propositions as Types
*Communications of the ACM, 58(12):75–84, December 2015.*

There’s also a video of Prof. Wadler presenting this material at *Strange Loop* in September 2015.

Code This

...see next slide.
Java has *subtyping*: a value of one type may be used at any more general type. So `String ⩽ Object`, and every `String` is an `Object`. This isn’t always straightforward.

```java
String[] a = { "Hello", "world" };  
Object[] b = a;  
b[0] = Boolean.FALSE;  
String s = a[0];  
System.out.println(s.toUpperCase());
```

1. Build a Java program around this.
2. Compile it.
3. Run it.
5. How might you change the Java language to prevent this?
What is Subtyping?

The idea of *behavioural subtyping* is that if \(S\) is a subtype of \(T\) then any \(S\) can be substituted in place of a \(T\).

Liskov’s principle of substitutivity:

\[
\text{\ldots properties that can be proved using the specification of an object’s presumed type should hold even though the object is actually a subtype of that type.}
\]

---

Barbara Liskov and Jeannette Wing

A Behavioral Notion of Subtyping

*ACM Transactions on Programming Languages and Systems* 16(6):1811–1841

DOI: 10.1145/197320.197383
Java has *subtyping*: a value of one type may be used at any more general type. So String $\leq$ Object, and every String is an Object. This isn’t always straightforward.

```java
String[] a = { "Hello", "world" }; // A small string array
Object[] b = a; // Now a and b are the same array
b[0] = Boolean.FALSE; // Drop in a Boolean object
String s = a[0]; // Oh, dear
System.out.println(s.toUpperCase()); // This isn’t going to be pretty
```

This compiles fine, with no errors or warnings.
Subtyping Arrays in Java

Java has subtyping: a value of one type may be used at any more general type. So `String ⩽ Object`, and every String is an Object. This isn’t always straightforward.

```
String[] a = { "Hello", "world" };       // A small string array
Object[] b = a;                          // Now a and b are the same array
b[0] = Boolean.FALSE;                    // Drop in a Boolean object
String s = a[0];                         // Oh, dear
System.out.println(s.toUpperCase());     // This isn’t going to be pretty
```

This compiles fine, with no errors or warnings.

When executed, we get a runtime type error at `b[0] = Boolean.FALSE`
Subtyping Arrays in Java

Java has **subtyping**: a value of one type may be used at any more general type. So String \(\leq\) Object, and every String is an Object. This isn’t always straightforward.

```java
String[] a = { "Hello", "world" };        // A small string array
Object[] b = a;                          // Now a and b are the same array
b[0] = Boolean.FALSE;                    // Drop in a Boolean object
String s = a[0];                         // Oh, dear
System.out.println(s.toUpperCase());     // This isn’t going to be pretty
```

This compiles fine, with no errors or warnings.

When executed, we get a runtime type error at `b[0] = Boolean.FALSE`

```java
Exception in thread "main" java.lang.ArrayStoreException: java.lang.Boolean at Subtype.main(Subtype.java:7)
```
Subtyping Arrays in Java

Java has *subtyping*: a value of one type may be used at any more general type. So `String \leq Object`, and every String is an Object. This isn’t always straightforward.

```java
String[] a = {"Hello", "world"};  // A small string array
Object[] b = a;                   // Now a and b are the same array
b[0] = Boolean.FALSE;             // Drop in a Boolean object
String s = a[0];                  // Oh, dear
System.out.println(s.toUpperCase()); // This isn’t going to be pretty
```

This compiles with no errors or warnings: in Java, if $S \leq T$ then $S[] \leq T[]$. 
Java has *subtyping*: a value of one type may be used at any more general type. So `String \leq Object`, and every String is an Object. This isn’t always straightforward.

```java
String[] a = { "Hello", "world" };  // A small string array
Object[] b = a;  // Now a and b are the same array
b[0] = Boolean.FALSE;  // Drop in a Boolean object
String s = a[0];  // Oh, dear
System.out.println(s.toUpperCase());  // This isn’t going to be pretty
```

This compiles with no errors or warnings: in Java, if $S \leq T$ then $S[] \leq T[]$.

That makes `String[] \leq Object[]`, and we can use a `String[]` anywhere we need an `Object[]`. 

Java has *subtyping*: a value of one type may be used at any more general type. So \( \text{String} \leq \text{Object} \), and every String is an Object. This isn’t always straightforward.

```java
String[] a = {"Hello", "world"}; // A small string array
Object[] b = a; // Now a and b are the same array
b[0] = Boolean.FALSE; // Drop in a Boolean object
String s = a[0]; // Oh, dear
System.out.println(s.toUpperCase()); // This isn’t going to be pretty
```

This compiles with no errors or warnings: in Java, if \( S \leq T \) then \( S[] \leq T[] \).

That makes \( \text{String[]} \leq \text{Object[]} \), and we can use a \( \text{String[]} \) anywhere we need an \( \text{Object[]} \).

Except that it isn’t and we can’t. So every array assignment gets a runtime check.
Subtyping Arrays in Java

Java has *subtyping*: a value of one type may be used at any more general type. So `String ⩽ Object`, and every String is an Object. This isn’t always straightforward.

```
String[] a = { "Hello", "world" };              // A small string array
Object[] b = a;                                // Now a and b are the same array
b[0] = Boolean.FALSE;                          // Drop in a Boolean object
String s = a[0];                               // Oh, dear
System.out.println(s.toUpperCase());           // This isn’t going to be pretty
```

What else could we do?
Java has *subtyping*: a value of one type may be used at any more general type. So String ⪯ Object, and every String is an Object. This isn’t always straightforward.

```java
String[] a = { "Hello", "world" };  // A small string array
Object[] b = a;  // Now a and b are the same array
b[0] = Boolean.FALSE;  // Drop in a Boolean object
String s = a[0];  // Oh, dear
System.out.println(s.toUpperCase());  // This isn’t going to be pretty
```

What else could we do?

- Prevent by-reference assignment, method call, and return. Only pass complete arrays.
Java has subtyping: a value of one type may be used at any more general type. So `String ⩽ Object`, and every `String` is an `Object`. This isn’t always straightforward.

```java
String[] a = { "Hello", "world" };          // A small string array
Object[] b = a;                             // Now a and b are the same array
b[0] = Boolean.FALSE;                       // Drop in a Boolean object
String s = a[0];                            // Oh, dear
System.out.println(s.toUpperCase());        // This isn’t going to be pretty
```

What else could we do?

- Prevent by-reference assignment, method call, and return. Only pass complete arrays.
- Forbid array update and make all arrays immutable.
Java has *subtyping*: a value of one type may be used at any more general type. So `String ⩽ Object`, and every String is an Object. This isn’t always straightforward.

```java
String[] a = { "Hello", "world" };           // A small string array
Object[] b = a;                             // Now a and b are the same array
b[0] = Boolean.FALSE;                       // Drop in a Boolean object
String s = a[0];                            // Oh, dear
System.out.println(s.toUpperCase());        // This isn’t going to be pretty
```

What else could we do?

- Prevent by-reference assignment, method call, and return. Only pass complete arrays.
- Forbid array update and make all arrays immutable.
- Remove array subtyping.
Subtyping Arrays in Java

Java has *subtyping*: a value of one type may be used at any more general type. So String \(\leq\) Object, and every String is an Object. This isn’t always straightforward.

```java
String[] a = { "Hello", "world" }; // A small string array
Object[] b = a; // Now a and b are the same array
b[0] = Boolean.FALSE; // Drop in a Boolean object
String s = a[0]; // Oh, dear
System.out.println(s.toUpperCase()); // This isn’t going to be pretty
```

What else could we do?

- Prevent by-reference assignment, method call, and return. Only pass complete arrays.
- Forbid array update and make all arrays immutable.
- Remove array subtyping.

Current Java keeps String[] \(\leq\) Object[] and inserts runtime type type checks.
Subtype variance

The issue here is that String[] is a parameterized type, like List<Object>, or in Haskell Maybe a and (a,b)→(b,a).

Suppose some type A⟨X⟩ depends on type X, and types S and T have S ≤ T. Then the dependency of A on X is:

- **Covariant** if A⟨S⟩ ≤ A⟨T⟩
- **Contravariant** if A⟨S⟩ ≥ A⟨T⟩
- **Invariant** if neither of these holds.

For example, in the Scala language, type parameters can be annotated with variance information: List[+T], Function[−S,+T]; while C# 4.0 introduced in and out variance tags.

In Java, arrays are typed as if they were covariant. But they aren’t.

see also *parameter covariance* in Eiffel
Typing in Object-Oriented languages

Ideally, a statically-checked object-oriented language should have a type system that is
Typing in Object-Oriented languages

Ideally, a statically-checked object-oriented language should have a type system that is

(a) usable, and
Typing in Object-Oriented languages

Ideally, a statically-checked object-oriented language should have a type system that is

(a) usable, and

(b) correct.
Typing in Object-Oriented languages

Ideally, a statically-checked object-oriented language should have a type system that is

(a) usable, and

(b) correct.

Building such type systems is a continuing challenge.
Typing in Object-Oriented languages

Ideally, a statically-checked object-oriented language should have a type system that is

(a) usable, and

(b) correct.

Building such type systems is a continuing challenge.

One problem is that subtyping is crucial to object-oriented programming, but unfortunately:
Typing in Object-Oriented languages

Ideally, a statically-checked object-oriented language should have a type system that is

(a) usable, and

(b) correct.

Building such type systems is a continuing challenge.

One problem is that subtyping is crucial to object-oriented programming, but unfortunately:

- subtyping is not inheritance; (really, it’s not)
  (although Java makes inheritance ⇒ subtyping)
Really, it's not


Typing in Object-Oriented languages

Ideally, a statically-checked object-oriented language should have a type system that is

(a) usable, and

(b) correct.

Building such type systems is a continuing challenge.

One problem is that subtyping is crucial to OO programming, but unfortunately:

- subtyping is not inheritance;  
  
  (really, it’s not)  
  
  (although Java makes inheritance ⇒ subtyping)
Typing in Object-Oriented languages

Ideally, a statically-checked object-oriented language should have a type system that is

(a) usable, and

(b) correct.

Building such type systems is a continuing challenge.

One problem is that subtyping is crucial to OO programming, but unfortunately:

- subtyping is not inheritance; (really, it’s not)
  (although Java makes inheritance ⇒ subtyping)

- it’s also extremely hard to get right.
Fixing object subtyping has been a busy research topic for several years.

You can see this by observing that the type declared for the max method in the Java collections class has gone from:

```java
public static Object max(Collection coll)
```

(Java 1.2, 1998)

...to...

```java
public static <T extends Object & Comparable<? super T>> T max(Collection<?> coll)
```

and it might still throw a ClassCastException.

(Java 8, 2014)

This is not a criticism: the new typing is more flexible, it saves on explicit downcasts, and the Java folks do know what they are doing.
Fixing object subtyping has been a busy research topic for several years.

You can see this by observing that the type declared for the max method in the Java collections class has gone from:

```java
public static Object max(Collection coll)
```

which always returns an Object, whatever is stored in the collection, to:

```java
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

This is not a criticism: the new typing is more flexible, it saves on explicit downcasts, and the Java folks do know what they are doing.
Fixing object subtyping has been a busy research topic for several years.

You can see this by observing that the type declared for the max method in the Java collections class has gone from:

```
public static Object max(Collection coll)
```

which always returns an Object, whatever is stored in the collection, to:

```
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```
How hard?

Fixing object subtyping has been a busy research topic for several years.

You can see this by observing that the type declared for the max method in the Java collections class has gone from:

```java
public static Object max(Collection coll)
```

which always returns an Object, whatever is stored in the collection, to:

```java
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

and it might still throw a ClassCastException.

(Java 1.2, 1998)
Fixing object subtyping has been a busy research topic for several years.

You can see this by observing that the type declared for the max method in the Java collections class has gone from:

```
public static Object max(Collection coll)
```

which always returns an Object, whatever is stored in the collection, to:

```
public static <T extends Object & Comparable<? super T>>
    T max(Collection<? extends T> coll)
```

and it might still throw a ClassCastException.

This is not a criticism: the new typing is more flexible, it saves on explicit downcasts, and the Java folks do know what they are doing.
Outline

1 Opening
2 Subtyping and Polymorphism
3 Beyond Hindley-Milner
4 System F
5 Datatypes
6 Beyond System F
7 Closing
## The Shootout

### The Benchmarks

<table>
<thead>
<tr>
<th>Ackermann's Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Access</td>
</tr>
<tr>
<td>Array Access II</td>
</tr>
<tr>
<td>Echo Client/Server</td>
</tr>
<tr>
<td>Exception Mechanisms</td>
</tr>
<tr>
<td>Fibonacci Numbers</td>
</tr>
<tr>
<td>Hash (Associative Array) Access</td>
</tr>
<tr>
<td>Hashes, Part II</td>
</tr>
<tr>
<td>Heapsort</td>
</tr>
<tr>
<td>List Operations</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
</tr>
</tbody>
</table>

### The Languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Implementation (Homepage)</th>
<th>Version (Download Page)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Awk</td>
<td>gawk</td>
<td>GNU Awk 3.0.6</td>
</tr>
<tr>
<td>Bash</td>
<td>bash</td>
<td>GNU sh, version 1.14.7(1)</td>
</tr>
<tr>
<td>C</td>
<td>gcc</td>
<td>egcs-2.91.66</td>
</tr>
<tr>
<td>C++</td>
<td>g++</td>
<td>egcs-2.91.66</td>
</tr>
<tr>
<td>Common Lisp</td>
<td>cmucl</td>
<td>CMU Common Lisp 18c</td>
</tr>
<tr>
<td>Eiffel</td>
<td>se</td>
<td>SmallEiffel The GNU Eiffel Compiler -- Release (~ 0.77) (patched to fix string append bug).</td>
</tr>
<tr>
<td>Emacs Lisp</td>
<td>xemacs</td>
<td>XEmacs 21.2 (beta37) &quot;Pan&quot; [Lucid] (i686-pc-linux)</td>
</tr>
<tr>
<td>Erlang</td>
<td>erlang</td>
<td>Erlang (BEAM) emulator version 5.0.1</td>
</tr>
</tbody>
</table>
### Computer Language Shootout Scorecard

**Which languages is best? Here's the Shootout Scorecard!**

The language with the most points is the winner! Now create your own!

<table>
<thead>
<tr>
<th>Language</th>
<th>Implementation</th>
<th>Score</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>gcc</td>
<td>767</td>
<td>0</td>
</tr>
<tr>
<td>C++</td>
<td>g++</td>
<td>763</td>
<td>0</td>
</tr>
<tr>
<td>Ocaml</td>
<td>ocaml</td>
<td>758</td>
<td>0</td>
</tr>
<tr>
<td>Java</td>
<td>java</td>
<td>673</td>
<td>0</td>
</tr>
<tr>
<td>Pike</td>
<td>pike</td>
<td>616</td>
<td>0</td>
</tr>
<tr>
<td>Lua</td>
<td>lua</td>
<td>539</td>
<td>2</td>
</tr>
<tr>
<td>Perl</td>
<td>perl</td>
<td>521</td>
<td>0</td>
</tr>
<tr>
<td>Common Lisp</td>
<td>cmuel</td>
<td>483</td>
<td>5</td>
</tr>
<tr>
<td>Ruby</td>
<td>ruby</td>
<td>439</td>
<td>0</td>
</tr>
<tr>
<td>Python</td>
<td>python</td>
<td>427</td>
<td>0</td>
</tr>
</tbody>
</table>

#### WEIGHS

<table>
<thead>
<tr>
<th>Test</th>
<th>Weight</th>
<th>Test</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ackermann's Function</td>
<td>1</td>
<td>Array Access</td>
<td>4</td>
</tr>
<tr>
<td>Array Access II</td>
<td>2</td>
<td>Echo Client/Server</td>
<td>5</td>
</tr>
<tr>
<td>Exceptions</td>
<td>1</td>
<td>Fibonacci Numbers</td>
<td>2</td>
</tr>
<tr>
<td>Hash (Associative Array) Access</td>
<td>1</td>
<td>Hashes Part II</td>
<td>4</td>
</tr>
<tr>
<td>Heapsort</td>
<td>4</td>
<td>List Processing</td>
<td>3</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>3</td>
<td>Method Calls</td>
<td>5</td>
</tr>
<tr>
<td>Statistical Moments</td>
<td>2</td>
<td>Nested Loops</td>
<td>4</td>
</tr>
<tr>
<td>Object Instantiation</td>
<td>5</td>
<td>Producer/Consumer Threads</td>
<td>3</td>
</tr>
<tr>
<td>Random Number Generator</td>
<td>3</td>
<td>Regular Expression Matching</td>
<td>4</td>
</tr>
<tr>
<td>Reverse a File</td>
<td>4</td>
<td>Sieve of Eratosthenes</td>
<td>4</td>
</tr>
<tr>
<td>Spell Checker</td>
<td>4</td>
<td>String Concatenation</td>
<td>2</td>
</tr>
</tbody>
</table>

**CPU Score Multiplier**: 1  
**Memory Score Multiplier**: 0
**Measurement is highly specific** -- the time taken for this benchmark task, by this toy program, with this programming language implementation, with these options, on this computer, with these workloads.

Same toy program, same computer, same workload -- but much slower.

Measurement is not prophesy.

<table>
<thead>
<tr>
<th>x86 Ubuntu™ Intel® Q6600® one core</th>
<th>x64 Ubuntu™ Intel® Q6600® quad-core</th>
<th>x86 Ubuntu™ Intel® Q6600® quad-core</th>
<th>x64 Ubuntu™ Intel® Q6600® one core</th>
</tr>
</thead>
<tbody>
<tr>
<td>C gcc</td>
<td>C gcc</td>
<td>C gcc</td>
<td>C gcc</td>
</tr>
<tr>
<td>Clojure</td>
<td>Clojure</td>
<td>Clojure</td>
<td>Clojure</td>
</tr>
<tr>
<td>C# Mono</td>
<td>C# Mono</td>
<td>C# Mono</td>
<td>C# Mono</td>
</tr>
<tr>
<td>C++ g++</td>
<td>C++ g++</td>
<td>C++ g++</td>
<td>C++ g++</td>
</tr>
</tbody>
</table>
Comparing Sort Algorithms

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

\[
\text{Sorter} = \forall \alpha. (\alpha \to \alpha \to \text{bool}) \to \text{list } \alpha \to \text{list } \alpha
\]

\[
\text{bubbleSorter, quickSorter, heapSorter, mergeSorter, bogoSorter, \ldots : Sorter}
\]

Here Sorter is a \textit{type scheme}, capturing the fact that 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.

\[
\text{simpleSorterTester =}
\lambda \text{sorter} . ((\text{sorter greaterThan} [5, 22, 2]) == [2, 5, 22])
\]

\[
\quad \quad \quad \quad \quad \text{and}((\text{sorter lessThan} [5, 22, 2]) == [22, 5, 2])
\]

\[
\quad \quad \quad \quad \quad \text{and}((\text{sorter dictionaryBefore} ["sort", "test"]) == ["sort", "test"])
\]

The \text{simpleSorterTester} takes a single polymorphic argument, the sorter, and uses it at multiple types. This is \textit{rank-2 polymorphism}. 

Comparing Sort Algorithms

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

```
simpleSorterTester : (∀α. (α → α → bool) → list α → list α) → bool
```

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

```
testManySorters = λsorters . λsorterTesters . (tabulatesorterTesters sorters)
testManySorters [bubbleSorter, quickSorter, heapSorter]
                [yourSorterTester, mySorterTester]
```

This is now beyond even rank-2 polymorphism, and we cannot manage without significantly more explicit type annotations.

```
testManySorters : list (∀α. (α → α → bool) → list α → list α) → list((∀α. (α → α → bool) → list α → list α) → bool) → list (list bool)
```
Outline

1. Opening
2. Subtyping and Polymorphism
3. Beyond Hindley-Milner
4. System F
5. Datatypes
6. Beyond System F
7. Closing
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:

\[
\text{identity} = \Lambda X.(\lambda x:X . x) : \forall X.(X \rightarrow X)
\]

With this definition:

\[ \text{identity} \ A \ M \ \xrightarrow{\beta} \ M \text{ for any } M : A. \]

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

\[ \text{identity} \ (\forall X.(X \rightarrow X)) \text{ identity} \ \xrightarrow{\beta} \ \text{identity}. \]

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 Metavariabes

In the last lecture Hindley-Milner types, type schemes and type variables were written with Greek letters ($\tau$, $\sigma$, $\alpha$). To distinguish System F this lecture moves to Roman letters for type (meta)variables.

### Notation

<table>
<thead>
<tr>
<th>Terms</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>Variables</td>
</tr>
<tr>
<td>$x, y, z$</td>
<td>$X, Y, Z$</td>
</tr>
<tr>
<td>Terms</td>
<td>Types</td>
</tr>
<tr>
<td>$M, N, \ldots$</td>
<td>$A, B, C, \ldots$</td>
</tr>
<tr>
<td>Term definitions,</td>
<td>Type definitions,</td>
</tr>
<tr>
<td>constants, constructors</td>
<td>constants, constructors</td>
</tr>
<tr>
<td>uncapsualisedwords</td>
<td>Product, Sum,</td>
</tr>
<tr>
<td></td>
<td>CapitalisedWords</td>
</tr>
</tbody>
</table>

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 $\Gamma = \{x_1 : A_1, x_2 : A_2, \ldots\}$. 

Ian Stark Advances in Programming Languages / Lecture 4: Higher Polymorphism 2016-09-30
## Rules for Types and Terms in System F

### Types

**Type Variable**

\[\Delta \vdash \text{Type } X \quad X \in \Delta\]

**Function Type**

\[\Delta \vdash \text{Type } A \quad \Delta \vdash \text{Type } B \quad \Delta \vdash \text{Type } A \to B\]

**For-All Type**

\[\Delta, X \vdash \text{Type } A \quad \Delta \vdash \text{Type } \forall X.A\]

### Terms

**Variable**

\[\Delta; \Gamma \vdash x : A \quad x : A \in \Gamma\]

**Abstraction**

\[\Delta; \Gamma, x : A \vdash M : B \quad \Delta; \Gamma \vdash (\lambda x : A. M) : A \to B\]

**Application**

\[\Delta; \Gamma \vdash F : A \to B \quad \Delta; \Gamma \vdash M : A \quad \Delta; \Gamma \vdash FM : B\]

**Type Abstraction**

\[\Delta, X; \Gamma \vdash M : A \quad \Delta; \Gamma \vdash \Lambda X. M : \forall X.A\]

**Type Application**

\[\Delta \vdash \text{Type } A \quad \Delta; \Gamma \vdash M : \forall X.A \quad \Delta; \Gamma \vdash MA : B[A/X]\]
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.

<table>
<thead>
<tr>
<th>Type Application</th>
<th>Term Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\Lambda X. M) A) (\rightarrow) (M{A/X})</td>
<td>((\lambda x: A. M) N) (\rightarrow) (M{N/x})</td>
</tr>
</tbody>
</table>

We write

\[ M \xrightarrow{\beta} N \]

...to indicate that term \(M\) reduces to \(N\) in zero or more steps of beta-reduction.
Some System F Types

Polymorphic functions now take explicit type arguments to indicate at what type they are being applied.

\[ \text{id}(\forall X. (X \rightarrow X)) \xrightarrow{\beta} \text{id} \]

This is enough to type all those sorters

\[
\begin{align*}
\text{Sorter} &= \forall X. (X \rightarrow X \rightarrow \text{Bool}) \rightarrow \text{List} X \rightarrow \text{List} X \\
\text{bubbleSorter}, \text{quickSorter}, \text{heapSorter}, \text{mergeSorter}, \text{bogoSorter}, \ldots : \text{Sorter}
\end{align*}
\]

\[
\begin{align*}
\text{SorterTester} &= \text{Sorter} \rightarrow \text{Bool} \\
&= (\forall X. (X \rightarrow X \rightarrow \text{Bool}) \rightarrow \text{List} X \rightarrow \text{List} X) \rightarrow \text{Bool} \\
\text{simpleSorterTester} : \text{SorterTester}
\end{align*}
\]

\[
\begin{align*}
\text{testManySorters} : (\text{List Sorter}) \rightarrow (\text{List SorterTester}) \rightarrow \text{List} (\text{List Bool})
\end{align*}
\]
Chris Okasaki
Even higher-order functions for parsing or Why would anyone ever want to use a sixth-order function?
DOI: 10.1017/S0956796898003001
Outline

1 Opening

2 Subtyping and Polymorphism

3 Beyond Hindley-Milner

4 System F

5 Datatypes

6 Beyond System F

7 Closing
Some Datatypes

<table>
<thead>
<tr>
<th>Basic Datatype Constructors</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Space</td>
<td>$\lambda x : A. M : A \rightarrow B$</td>
<td>As in functions, lambdas, procedures, methods,…</td>
</tr>
<tr>
<td>Product</td>
<td>$(M, N) : A \times B$</td>
<td>As in product, record, struct,…</td>
</tr>
<tr>
<td>Sum</td>
<td>$\text{inl, inr} : A + B$</td>
<td>As in sum, variant, union,…</td>
</tr>
</tbody>
</table>

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**

\[
\text{Prod } X \ Y = \forall Z.((X \to Y \to Z) \to Z)
\]

\[
\text{pair } = \lambda X.\lambda Y. (\lambda x:X.\lambda y:Y. (\lambda Z.\lambda f:(X \to Y \to Z).(f x y)))
\]

\[
\text{pair } : \forall X.\forall Y.(X \to Y \to \text{Prod } X \ Y)
\]

\[
\text{pair } A \ B \ M \ N \xrightarrow{\beta} \lambda Z.\lambda f:(A \to B \to Z).f M N
\]

**First Projection**

\[
\text{fst } = \lambda X.\lambda Y.\lambda p:(\text{Prod } X \ Y). p X (\lambda x:X.\lambda y:Y.x)
\]

\[
\text{fst } : \forall X.\forall Y.\text{Prod } X \ Y \to X
\]

\[
\text{fst } A \ B \ (\text{pair } A \ B \ M \ N) \xrightarrow{\beta} M
\]

(Earlier versions of this slide gave \(\text{fst}\) incorrectly)

**Syntactic Sugar**

\[
A \times B = \text{Prod } A \ B
\]

\[
(M, N)_{A,B} = \text{pair } A \ B \ M \ N : A \times B
\]

\[
\text{fst}_{A,B} = \text{fst } A \ B : A \times B \to A
\]
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

\[ \text{snd} \ A \ B \ (\text{pair} \ A \ B \ M \ N) \xrightarrow{\beta} N \]

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

\[ \text{Sum} \ X \ Y = \forall Z. ((X \to Z) \to (Y \to Z) \to Z) \]

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

1. Opening
2. Subtyping and Polymorphism
3. Beyond Hindley-Milner
4. System F
5. Datatypes
6. Beyond System F
7. 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.
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 $F_{\langle F \rangle}$ (F-sub) introduces bounded quantification $\forall X<A. B$.

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

With System $F_{\omega}$ we get abstraction over type operators of higher kinds; for example:

$$(\lambda (X: \ast). (\lambda (Y: \ast). (X \times Y \times Y))): \ast \to \ast \to \ast$$

And the existential type $\exists X. A$ is dual to the universal $\forall X. A$, but can be encoded using it...
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 $F_\lt$: (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>> ...`
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 $F_\prec$: (F-sub) introduces \textit{bounded quantification} $\forall X<A.B$.

Java uses this in declarations like \texttt{class A<T extends String> ...}

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

Java uses this too, in \texttt{class A<T extends Comparable<T>> ...}

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

$$(\lambda(X:*) \cdot \lambda(Y:*)(X \times Y \times Y)) : * \to * \to *.$$
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 $F_\prec$:* (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 $F(\cdot)$.

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 $F_\omega$* we get abstraction over type operators of higher kinds; for example:

$$(\lambda (F:(* \to * \to *)).\lambda (X:*)(F \times X)) : (* \to * \to *) \to * \to * .$$
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 $F_\prec$: (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:

$$(\lambda(X:*).\lambda(Y:*).(X \times Y \times Y)) : * \rightarrow * \rightarrow * .$$

With System $F_\omega$ we get abstraction over type operators of higher kinds; for example:

$$(\lambda(F:(* \rightarrow * \rightarrow *)).\lambda(X:*).(F X X)) : (* \rightarrow * \rightarrow *) \rightarrow * \rightarrow * .$$

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

1. Read This

Information about the APL written coursework appears today on the course website. The assignment is to investigate one of five programming-language topics and write a 10-page report on it.

During next Tuesday’s lecture I shall give a brief overview of each topic, and explain more about what’s involved in the assignment. Before then, download and read the assignment handout on the course website.

2. Do This

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

Extension

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