Advances in Programming Languages

Lecture 4: Higher Polymorphism

Ian Stark

School of Informatics
The University of Edinburgh

Friday 30 September 2016
Semester 1 Week 2

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
- 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

Outline

Review

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

Java	Set<String>	constructor Set	parameter String
Haskell	Tree int	constructor Tree	parameter int
OCaml	(bool −> bool) list	constructor list	parameter bool−>bool

Review

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

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

Java generics

```
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

Review

Hindley-Milner types, *type schemes* σ 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.

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

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

Outline

Homework (1/2)

Read This

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

http://is.gd/wadler_pat

There's also a video of Prof. Wadler presenting this material at *Strange Loop* in September 2015. https://is.gd/wadler_pat_video

Code This

. . . see next slide.

Homework (2/2)

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

```
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.
4. What happens? Can you explain why?
5. How might you change the Java language to prevent this?

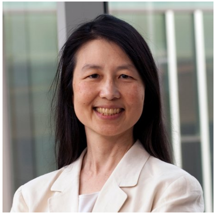
What is Subtyping?

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:

> *. . . 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
2008 Turing Award



Jeannette Wing
VP Microsoft Research

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

Subtyping Arrays in Java

Java has *subtyping*: a value of one type may be used at any more general type. So String $\leqslant$ 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.

Subtyping Arrays in Java

Java has *subtyping*: a value of one type may be used at any more general type. So String $\leqslant$ 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 $\leqslant$ 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

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

Subtyping Arrays in Java

Java has *subtyping*: a value of one type may be used at any more general type. So String $\leqslant$ 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 with no errors or warnings: in Java, if S $\leqslant$ T then S[] $\leqslant$ T[].

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

Subtyping Arrays in Java

Java has *subtyping*: a value of one type may be used at any more general type. So String $\leqslant$ 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 with no errors or warnings: in Java, if S $\leqslant$ T then S[] $\leqslant$ T[].

That makes String[] $\leqslant$ Object[], and we can use a String[] anywhere we need an 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 $\leqslant$ 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?

Subtyping Arrays in Java

Java has *subtyping*: a value of one type may be used at any more general type. So String $\leqslant$ 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?

- Prevent by-reference assignment, method call, and return. Only pass complete arrays.

Subtyping Arrays in Java

Java has *subtyping*: a value of one type may be used at any more general type. So String $\leqslant$ 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?

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

Subtyping Arrays in Java

Java has *subtyping*: a value of one type may be used at any more general type. So String $\leqslant$ 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?

- 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 $\leqslant$ 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?

- 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[] $\leqslant$ Object[] and inserts runtime 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 $A\langle X\rangle$ depends on type X, and types S and T have $S \leqslant T$. Then the dependency of A on X is:

Covariant	if $A\langle S\rangle \leqslant A\langle T\rangle$	e.g. pair $A\langle X\rangle = X * X$
Contravariant	if $A\langle S\rangle \geqslant A\langle T\rangle$	e.g. test $A\langle X\rangle = (X \rightarrow \text{bool})$
Invariant	if neither of these holds.	e.g. array $A\langle X\rangle = X[]$

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 $\Rightarrow$ subtyping)

Really, it's not

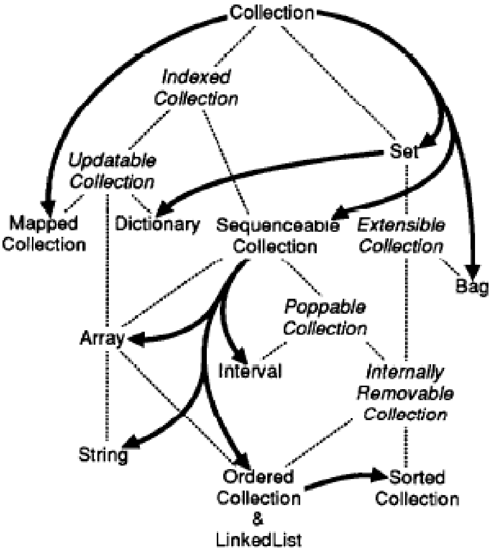


Figure 5: Interfaces versus Inheritance

W. R. Cook.
Interfaces and specifications for the Smalltalk-80 collection classes.
Proc. OOPSLA '92, pp. 1–15.

W. R. Cook, W. Hill, and P. S. Canning.
Inheritance is not subtyping.
Proc. POPL '90, pp. 125–135.

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 $\Rightarrow$ 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 $\Rightarrow$ subtyping)
- it's also extremely hard to get right.

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 1.2, 1998)

```
public static Object max(Collection 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 1.2, 1998)

public static Object max(Collection coll)

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

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 1.2, 1998)

> **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 1.2, 1998)

```
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. (Java 8, 2014)

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 1.2, 1998)

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. (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.

Outline

The Great Computer Language Shootout

[384 out of 576 benchmark programs completed]

The Shootout

The Benchmarks
Ackermann's Function
Array Access
Array Access II
Echo Client/Server
Exception Mechanisms
Fibonacci Numbers
Hash (Associative Array) Access
Hashes, Part II
Heapsort
List Operations
Matrix Multiplication

	Language	Imple-mentation (Homepage)	Version (Download Page)
1.	Awk	gawk	GNU Awk 3.0.6
2.	Bash	bash	GNU sh, version 1.14.7(1)
3.	C	gcc	egcs-2.91.66
4.	C++	g++	egcs-2.91.66
5.	Common Lisp	cmucl	CMU Common Lisp 18c
6.	Eiffel	se	SmallEiffel The GNU Eiffel Compiler -- Release (- 0.77) (patched to fix string append bug).
7.	Emacs Lisp	xemacs	XEmacs 21.2 (beta37) "Pan" [Lucid] (i686-pc-linux)
8.	Erlang	erlang	Erlang (BEAM) emulator version 5.0.1

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!

Recalculate Scores Reset

CPU Score Multiplier 1 Memory Score Multiplier 0

WEIGHTS

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

SCORES

Language	Implementation	Score	Missing
C	gcc	767	0
C++	g++	763	0
Ocaml	ocaml	758	0
Java	java	673	0
Pike	pike	616	0
Lua	lua	539	2
Perl	perl	521	0
Common Lisp	cmucl	483	5
Ruby	ruby	439	0
Python	python	427	0

The Computer Language Benchmarks Game [[Play]]

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.

x86 Ubuntu™ Intel® Q6600® one core	x64 Ubuntu™ Intel® Q6600® quad-core	x86 Ubuntu™ Intel® Q6600® quad-core	x64 Ubuntu™ Intel® Q6600® one core
Ada 2005 GNAT	Ada 2005 GNAT	Ada 2005 GNAT	Ada 2005 GNAT
C gcc	C gcc	C gcc	C gcc
Clojure	Clojure	Clojure	Clojure
C# Mono	C# Mono	C# Mono	C# Mono
C++ g++	C++ g++	C++ g++	C++ g++

Comparing Sort Algorithms

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

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

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

Here Sorter is a *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.

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

The simpleSorterTester 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.

$$\mathsf{simpleSorterTester} \quad : \quad (\forall\alpha.(\alpha \to \alpha \to \mathsf{bool}) \to \mathsf{list}\,\alpha \to \mathsf{list}\,\alpha) \to \mathsf{bool}$$

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

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

$$\begin{aligned}\mathsf{testManySorters}\;&[\mathsf{bubbleSorter},\ \mathsf{quickSorter},\ \mathsf{heapSorter}]\\ &[\mathsf{yourSorterTester},\ \mathsf{mySorterTester}]\end{aligned}$$

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

$$\mathsf{testManySorters} \quad : \quad \mathsf{list}(\forall\alpha.(\alpha \to \alpha \to \mathsf{bool}) \to \mathsf{list}\,\alpha \to \mathsf{list}\,\alpha) \to \mathsf{list}((\forall\alpha.(\alpha \to \alpha \to \mathsf{bool}) \to \mathsf{list}\,\alpha \to \mathsf{list}\,\alpha) \to \mathsf{bool}) \to \mathsf{list}\,(\mathsf{list}\,\mathsf{bool})$$

Outline

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:

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

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

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

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

In the last lecture Hindley-Milner types, 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

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

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.

$$\mathsf{identity}\ (\forall X.(X \to X))\ \mathsf{identity} \xrightarrow{\beta} \mathsf{identity}\ .$$

This is enough to type all those sorters

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

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

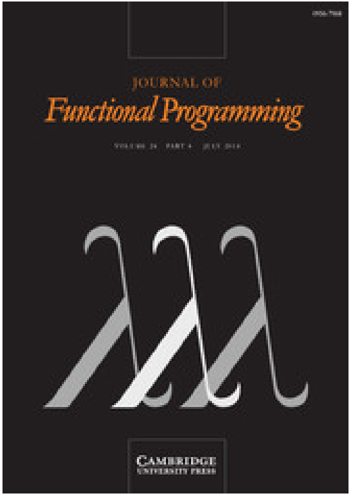
$$\begin{aligned} \mathsf{SorterTester} &= \mathsf{Sorter} \to \mathsf{Bool} \\ &= (\forall X.(X \to X \to \mathsf{Bool}) \to \mathsf{List}\,X \to \mathsf{List}\,X) \to \mathsf{Bool} \end{aligned}$$

$$\mathsf{simpleSorterTester} : \mathsf{SorterTester}$$

$$\mathsf{testManySorters} : (\mathsf{List}\,\mathsf{Sorter}) \to (\mathsf{List}\,\mathsf{SorterTester}) \to \mathsf{List}(\mathsf{List}\,\mathsf{Bool})$$

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

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	$\mathsf{inl}, \mathsf{inr} : A + B$	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

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

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

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

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

First Projection

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

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

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

Syntactic Sugar

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

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

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

(Earlier versions of this slide gave fst incorrectly)

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

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

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

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>> ...

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:

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

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:

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

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

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 }.