Advances in Programming Languages

Lecture 5: Higher Polymorphism

Ian Stark

School of Informatics
The University of Edinburgh

Monday 1 October 2018
Semester 1 Week 3

THE UNIVERSITY *of* EDINBURGH

https://wp.inf.ed.ac.uk/apl18

PLInG: Programming Language Interest Group

School of Informatics

Next Meeting: 1pm Thursday 4 October 2018 in IF 3.02

PLInG is an informal meeting series for anyone interested in programming languages. All Informatics staff and students are welcome to participate. The group meets every two weeks or so, with presentation of interesting recent papers, discussion of work in progress, informal talks by visitors, etc.

See also: SPLS, PLUG, other-PL-based initialisms, . . .
I'll post more information on the apl-students mailing list and elsewhere.

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

Homework

Before the next lecture find an online tutorial for each of the assignment topics:

- Parallel performance portability with Lift
- Dynamic information flow policies in Jeeves
- Programming quantum computation with Quipper
- Query expressions for language-integrated database access in F#
- Probabilistic programming for statistical inference in Stan

Send me your list of five links and I will summarize them for the class.

Use them to help choose your topic.

"You can never understand one language until you understand at least two."

Ronald Searle, artist (1920–2011)

Review

Types appear widely in programming languages, used for many purposes, and play a significant role in the organisation and structuring of code.

The lambda calculus is a model of computation that takes functions as fundamental, and builds everything out of variables, function abstraction, and function application. Formal rules give ways to build terms, reduce one term to another, and assign types to terms.

Many programming languages include the facility for constructing and using functions as first-class citizens alongside other sorts of data. Closures combine function bodies with variable environments, and together with higher-order functions these provide a powerful programming abstraction.

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.

Ad-hoc Polymorphism in Java

```
Shape[] shapeArray;
...
for (Shape s : shapeArray)    // For every shape in the array ...
{ s.draw(); }                 // ... invoke its "draw" method.
```

Parametric Polymorphism in OCaml

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

Review

Parametric polymorphism is one kind of *generic* programming, and parametric polymorphism in object-oriented languages is often known as *generics*.

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) { ... }
```

Outline

Previous Homework

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" };        // Array of strings
Object[] b = a;                           // Array of objects (every string is an object)
b[0] = Boolean.FALSE;                     // Assign object to array of objects
String s = a[0];                          // Fetch string from array of strings
System.out.println(s.toUpperCase());      // Convert string to upper-case
```

1. Build a Java program around this.
2. Compile it. Run it.
3. What happens, and when? Can you explain why?
4. How might you change the Java language to prevent this?
5. Pick another object-oriented language: what happens when you try this there?

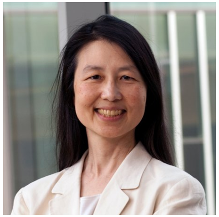
What is Subtyping?

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.

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[].

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?

- 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

(a) usable, and

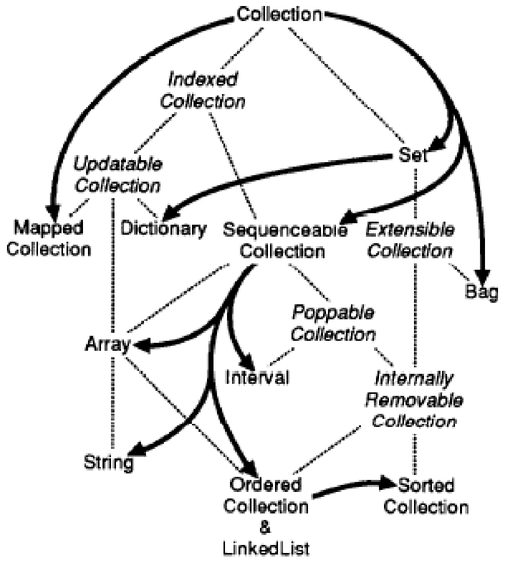
(b) correct.

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



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)
- 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)
```

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 11, 2018)

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

Remember the Lambda Calculus?

All sorts of interesting types can be added to the basic typed lambda calculus. Many can in fact be encoded in one way or another, but it's often convenient to have them presented explicitly. For example, here is one representation for *pairs* (M_1, M_2) with *product type* $(\tau_1 \times \tau_2)$.

Terms

$$\frac{\text{Term } M_1 \quad \text{Term } M_2}{\text{Term } (M_1, M_2)}$$

$$\frac{}{\text{Term fst}}$$

$$\frac{}{\text{Term snd}}$$

Typing

$$\frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2}$$

$$\frac{}{\vdash \mathsf{fst} : (\tau_1 \times \tau_2) \to \tau_1}$$

$$\frac{}{\vdash \mathsf{snd} : (\tau_1 \times \tau_2) \to \tau_2}$$

Reduction

$$\mathsf{fst}(M_1, M_2) \longrightarrow M_1$$

$$\mathsf{snd}(M_1, M_2) \longrightarrow M_2$$

This can be extended to *tuples* of arbitrary size $(M_1, M_2, \ldots, M_n)$, and there are similar rules for sum types $(\tau_1 + \tau_2)$.

Exercise: Write sum type rules

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

$$\frac{\Gamma \vdash M_1 : \sigma_1 \quad \Gamma, x : \sigma_1 \vdash M_2 : \sigma_2}{\Gamma \vdash \mathsf{let}\, x = M_1 \,\mathsf{in}\, M_2 : \sigma_2}$$

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 that is:

- Correct — it can be checked using the rules; and
- The *most general* type — all other possible 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).

Outline

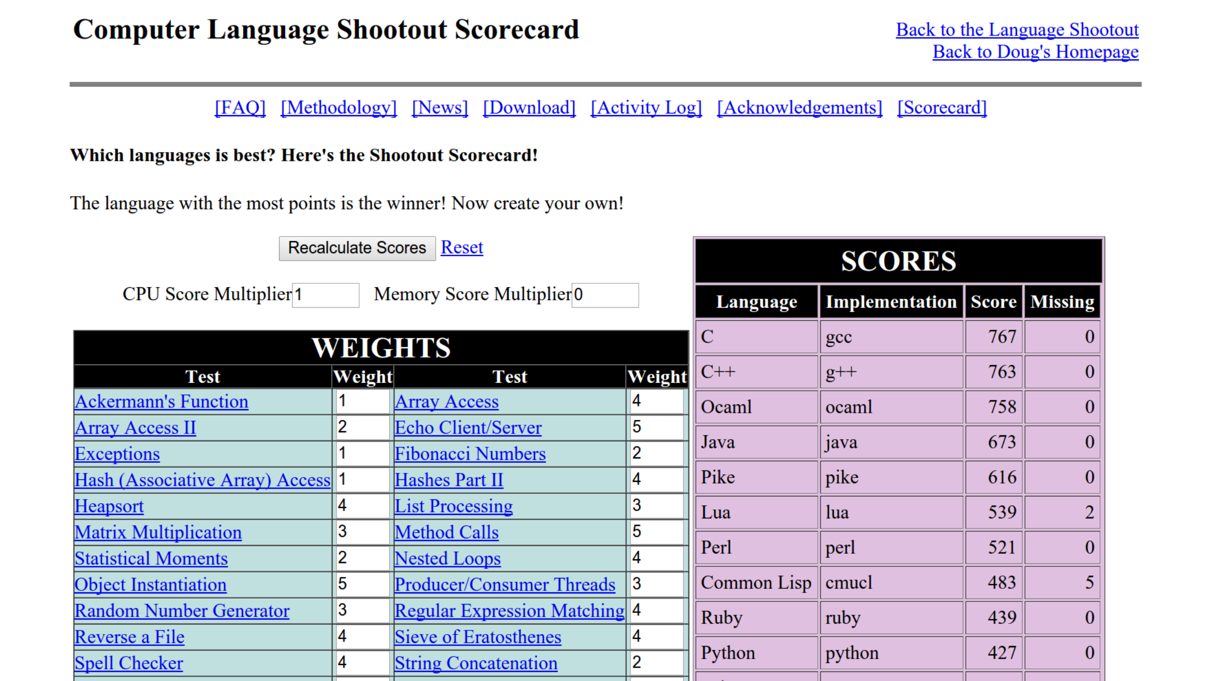
The Great Computer Language Shootout

[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

The Languages			
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

Back to the Language Shootout
Back to Doug's Homepage

[FAQ] [Methodology] [News] [Download] [Activity Log] [Acknowledgements] [Scorecard]

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

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

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?

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

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

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

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