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.
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 Hindley-Milner

4 Beyond Hindley-Milner

5 Closing
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)
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**

<table>
<thead>
<tr>
<th>Language</th>
<th>Type Expression</th>
<th>Constructor</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Set&lt;String&gt;</td>
<td>Set</td>
<td>String</td>
</tr>
<tr>
<td>Haskell</td>
<td>Tree int</td>
<td>Tree</td>
<td>int</td>
</tr>
<tr>
<td>OCaml</td>
<td>(bool -&gt; bool) list</td>
<td>list</td>
<td>bool-&gt;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.

Ad-hoc Polymorphism in Java

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

Parametric Polymorphism in OCaml

```ocaml
let rec map f = function ...

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```
Parametric polymorphism is one kind of generic programming, and parametric polymorphism in object-oriented languages is often known as generics.

### 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) { ... }
```
Outline

1. Opening
2. Subtyping and Polymorphism
3. Hindley-Milner
4. Beyond Hindley-Milner
5. Closing
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" }; // 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 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 \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`

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

This compiles with no errors or warnings: in Java, if `S ⩽ T` then `S[] ⩽ T[]`.

That makes `String[] ⩽ 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 \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 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 $S \leq T$. Then the dependency of $A$ on $X$ is:

- **Covariant** if $A\langle S\rangle \leq A\langle T\rangle$
  
  e.g. pair $A\langle X\rangle = X \times X$

- **Contravariant** if $A\langle S\rangle \geq A\langle T\rangle$

  e.g. test $A\langle X\rangle = (X \to bool)$

- **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 ⇒ 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)

- 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)
```

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.

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 Hindley-Milner

4 Beyond Hindley-Milner

5 Closing
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 \((\tau_1 \times \tau_2)\).

<table>
<thead>
<tr>
<th>Terms</th>
<th>Typing</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{Term } M_1) \quad \text{Term } M_2</td>
<td>(\Gamma \vdash M_1 : \tau_1) \quad (\Gamma \vdash M_2 : \tau_2)</td>
<td>(\text{fst}(M_1, M_2) \rightarrow M_1)</td>
</tr>
<tr>
<td>(\text{Term } (M_1, M_2))</td>
<td>(\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2)</td>
<td>(\text{snd}(M_1, M_2) \rightarrow M_2)</td>
</tr>
<tr>
<td>(\text{Term } \text{fst})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\vdash \text{fst} : (\tau_1 \times \tau_2) \rightarrow \tau_1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\vdash \text{snd} : (\tau_1 \times \tau_2) \rightarrow \tau_2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\text{fst : } \forall \alpha, \beta. (\alpha \times \beta) \rightarrow \alpha
\]

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

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

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

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

Generalise types \(\tau\) to type schemes \(\sigma\) which quantify over type variables \(\alpha, \beta, \gamma, \ldots\)

\[
\sigma ::= \tau \mid \forall \alpha. \sigma
\]

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

Concrete types \(\tau\) are instances of a type scheme \(\sigma\).

Also sometimes referred to as polytype \(\sigma\) and monotype \(\tau\).
Checking Polymorphic Types

Rules for Polymorphic Types

Generalize

\[ \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall \alpha. \sigma} \quad \alpha \notin \text{free}(\Gamma) \]

Specialize

\[ \frac{\Gamma \vdash M : \forall \alpha. \sigma}{\Gamma \vdash M : \sigma\{\tau/\alpha\}} \quad \text{free}(\sigma) \cap \text{free}(\tau) = \emptyset \]

Let-binding

\[ \frac{\Gamma \vdash M_1 : \sigma_1 \quad \Gamma, x : \sigma_1 \vdash M_2 : \sigma_2}{\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : \sigma_2} \]

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

\[ \text{let } x = M \text{ in } N \]

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:

\[
\text{let } x = M \text{ in } N
\]

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

1. Opening
2. Subtyping and Polymorphism
3. Hindley-Milner
4. Beyond Hindley-Milner
5. Closing
The Great Computer Language Shootout

[384 out of 576 benchmark programs completed]

The Shootout

<table>
<thead>
<tr>
<th>The Benchmarks</th>
<th>The Languages</th>
<th>Version (Download Page)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ackermann's Function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array Access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array Access II</td>
<td>1. Awk gawk GNU Awk 3.0.6</td>
<td></td>
</tr>
<tr>
<td>Echo Client/Server</td>
<td>2. Bash bash GNU sh, version 1.14.7(1)</td>
<td></td>
</tr>
<tr>
<td>Exception Mechanisms</td>
<td>3. C gcc egcs-2.91.66</td>
<td></td>
</tr>
<tr>
<td>Fibonacci Numbers</td>
<td>4. C++ g++ egcs-2.91.66</td>
<td></td>
</tr>
<tr>
<td>Hash (Associative Array) Access</td>
<td>5. Common Lisp cmuecl CMU Common Lisp 18c</td>
<td></td>
</tr>
<tr>
<td>Hashes, Part II</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heapsort</td>
<td>6. Eiffel se SmallEiffel The GNU Eiffel Compiler -- Release (- 0.77) (patched to fix string append bug).</td>
<td></td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>8. Erlang erlang Erlang (BEAM) emulator version 5.0.1</td>
<td></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!

### Recalculate Scores Reset

CPU Score Multiplier: 1 Memory Score Multiplier: 0

<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>

### SCORES

<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>emucl</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>
**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></th>
<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>
<td>C gcc</td>
</tr>
<tr>
<td>Clojure</td>
<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>
<td>C# Mono</td>
</tr>
<tr>
<td>C++ g++</td>
<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.

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

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

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.

\[
\text{simpleSorterTester} = \\
\lambda \text{sorter}. (((\text{sorter greaterThan } [5, 22, 2]) \equiv [2, 5, 22])
\text{ and}((\text{sorter lessThan } [5, 22, 2]) \equiv [22, 5, 2])
\text{ and}((\text{sorter dictionaryBefore } ["sort", "test"]) \equiv ["sort", "test"]
\]

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.

\[ \text{simpleSorterTester} : (\forall \alpha. (\alpha \rightarrow \alpha \rightarrow \text{bool}) \rightarrow \text{list } \alpha \rightarrow \text{list } \alpha) \rightarrow \text{bool} \]

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

\[ \text{testManySorters} = \lambda \text{sorters} . \lambda \text{sorterTesters} . \left( \text{tabulatesorterTesters sorters} \right) \]

\[ \text{testManySorters} \begin{bmatrix} \text{bubbleSorter}, \text{quickSorter}, \text{heapSorter} \\ \text{yourSorterTester}, \text{mySorterTester} \end{bmatrix} \]

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

\[ \text{testManySorters} : \text{list}(\forall \alpha. (\alpha \rightarrow \alpha \rightarrow \text{bool}) \rightarrow \text{list } \alpha \rightarrow \text{list } \alpha) \rightarrow \text{list}(\forall \alpha. (\alpha \rightarrow \alpha \rightarrow \text{bool}) \rightarrow \text{list } \alpha \rightarrow \text{list } \alpha) \rightarrow \text{bool} \rightarrow \text{list (list bool)} \]
Outline

1 Opening

2 Subtyping and Polymorphism

3 Hindley-Milner

4 Beyond Hindley-Milner

5 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