

Advances in Programming Languages

Parameterized Types and Polymorphism

Ian Stark

School of Informatics
The University of Edinburgh

Monday 24 September 2018
Semester 1 Week 2



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

- 1 Opening
- 2 First-Class Functions
- 3 Parameterized Types
- 4 Polymorphism
- 5 Closing

Outline

- 1 Opening
- 2 First-Class Functions
- 3 Parameterized Types
- 4 Polymorphism
- 5 Closing

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.

“Language is a cracked kettle on which we beat out tunes for bears to dance to, while all the time we long to move the stars to pity.”
Gustave Flaubert (1821–1880)

Homework

Watch This

https://is.gd/yang_vision (Video, 2m18s)

Language design and software verification, MIT Technology Review, May 2017


Jean Yang, Assistant Professor of Computer Science, Carnegie Mellon University



Do This

Find out about the *Blub Paradox* and read the article that named it.

Read This

 **Achim Jung**
A Short Introduction to the Lambda Calculus

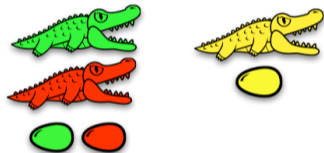
http://is.gd/jung_lc

You may find it helpful to read pages 1–7 of Pierce’s “Foundational Calculi for Programming Languages” alongside, although that’s wholly untyped.

http://is.gd/pierce_fc

After Jung's technical paper try these two, very different, pages:

- Bret Victor's [Alligator Eggs](#)
(Try Takashi Yamamiya's [animation](#) of these)
- Wikipedia on [First-Class Functions](#)
(If you like that, dip into the opinions on the [Talk](#) page)



Outline

- 1 Opening
- 2 First-Class Functions**
- 3 Parameterized Types
- 4 Polymorphism
- 5 Closing

First-Class Functions

The lambda-calculus originated as mathematical model for describing possible computation; programming languages arose as a vehicle for carrying it out. They have some common history, but also notable differences. One is the treatment of functions.

The lambda-calculus is built of functions: they make up both program and data. Many programming languages do include functions — as procedures, methods, etc. — but usually as specialised control structures where the functions themselves are not values in the language.

The distinctive feature here is to make functions *first-class* in a language: passed as arguments to other functions, returned as results, created anonymously during execution, stored, combined, applied, and discarded.

These make a powerful abstraction in programming. For example, first-class and higher-order functions can replace many uses of introspection and runtime code generation, while being fully compiled and statically checkable.

Examples

A key marker for first-class functions in a language is the availability of lambda-abstraction to create *anonymous functions* (or *function literals* or simply *lambdas*).

LISP languages — including Common Lisp, Scheme, Racket — have always included lambdas and higher-order functions.

```
(lambda (x y) (+ x y)) ; Add two numbers
```

The same is true for other functional languages, such as those based on ML: Standard ML, OCaml, F#:

```
(fn p => 2*p)          (* Double a value, Standard ML syntax *)
```

as well as Haskell:

```
\p -> \q -> p ++ q    -- Concatenate two lists.
```

Examples

Java

(since Java 8)

$q \rightarrow q+1$

$(a,b) \rightarrow \text{Math.sqrt}(a*a + b*b)$

$(\text{String } s, \text{String } t) \rightarrow \{ \text{String result} = s + t; \textbf{return} \text{ result}; \}$

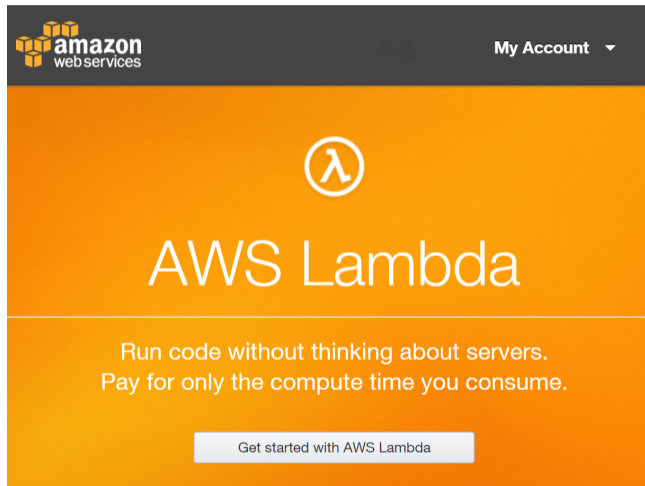
Smalltalk

$[:x \mid x*x*x]$

Scala

$(f: \text{String} \Rightarrow \text{Int}, s: \text{String}) \Rightarrow f(s)$

See Wikipedia “Anonymous functions” for many, many more



amazon
web services

My Account ▾

Λ

AWS Lambda

Run code without thinking about servers.
Pay for only the compute time you consume.

Get started with AWS Lambda

Upload your code as a lambda. Run it when needed. No server management, runs in parallel if called multiple times, scales with use.

First 1 million calls are free

After that, \$0.0000002 each

<https://aws.amazon.com/lambda>

Closures

When a function builds another function, that may include references to the original function's arguments. In lambda-calculus, for example:

$$(\lambda x:\text{num}. (\lambda y:\text{num}. (x * y))) 5 \longrightarrow \lambda y:\text{num}. (5 * y)$$

In a programming language this is typically implemented by returning not just a function but also the values of its free variables:

$$\{x = 5; (\lambda y:\text{num}. x * y)\}.$$

This combination of function and variable environment is known as a *closure*.

Closures are particularly significant for imperative languages, where they may require extending the lifetime of local variables.

Java, for example, disallows closures that refer to mutable state.

Forbidden Closure in Java

```
import java.util.function.Predicate; // A "predicate" is a test that takes
                                     // a value and returns a boolean result.

                                     // This checker takes a string predicate and sees
public class Checker {               // what answer it returns for one specific value

    public void runCheck(Predicate<String> p) {
        if (p.test("secret"))
            { System.out.println("Pass"); }
        else
            { System.out.println("Fail"); }
        }
    }
```

Forbidden Closure in Java

```
...
public void runLengthTests (Checker c) { // Given a checker, try it out on some predicates
    int n = 0;

    Predicate<String> lengthTest = // A predicate to select strings of n or more characters
        (String y) -> {
            System.out.println("Comparing to: "+n);
            return (y.length() >= n);
        };

    n = 4; c.runCheck(lengthTest); // Run the checker to test length 4 or more

    n = 8; c.runCheck(lengthTest); // Run the checker to test length 8 or more
}

// What happens when we try to compile this?
```


Forbidden Closure in Java

```
...
public void runLengthTests (Checker c) { // Given a checker, try it out on some predicates
    int n = 0;

    Predicate<String> lengthTest = // A predicate to select strings of n or more characters
        (String y) -> {
            System.out.println("Comparing to: "+n); // error when compiling
            return (y.length() >= n); // error when compiling
        };

    n = 4; c.runCheck(lengthTest); // Run the checker to test length 4 or more

    n = 8; c.runCheck(lengthTest); // Run the checker to test length 8 or more
}

// error: local variables referenced from a lambda expression must be final or effectively final
```

Outline

- 1 Opening
- 2 First-Class Functions
- 3 Parameterized Types**
- 4 Polymorphism
- 5 Closing

Java is serious about abstraction

Java works almost entirely by class-based object-oriented programming; it encourages the use of abstract classes through inheritance and interfaces; and it does not expose the private workings of classes and packages.

Java is serious about typing

Java has strong static typing: all programs are checked for type-correctness at compile-time. Bytecode is checked again when classes are loaded, by the *bytecode verifier*, before execution. Even the *invokedynamic* bytecode introduced in Java 7 checks its dynamically created code.

All this means that for any feature or programming technique in Java there is a very strong drive to have it properly handled within the type system.

Java Arrays

Java has built-in arrays of fixed size containing elements of a given type.

```
float [], String [], Object [], TimeStamp []
```

Arrays in Java are a *parameterized* type: there are many different sorts of array, depending on the type of value they contain.

Arrays and Collections

Java Collections

The `java.util` package contains implementations of many kinds of collection — sets, lists, queues, maps, etc.

Before Java 5, these all contained any sort of object, or mixture of objects.

```
// Implementations of some collections  
public class HashSet implements Set  
public class ArrayList implements List
```

```
// Methods in List interface  
public void add(int index, Object o)  
public Object get(int index)
```

As any object can be treated as having class `Object`, this is correct, but very approximate.

In particular, fetching values out of a collection often requires *downcasting* from `Object` to a more specific type before doing any further computation. That involves a runtime check, and that might fail.

Java Generics



Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler

Making the Future Safe for the Past: Adding Genericity to the Java Programming Language

In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 183–200. ACM Press, 1998.

DOI: 10.1145/286942.286957

```
// Implementations of some collections
.. HashSet<E> implements Set<E>
.. ArrayList<E> implements List<E>
```

```
// Methods in List<E> interface
public void add(int index, E element)
public E get(int index)
```

Retrofitting this to the language was a major challenge, developed over several years: generic code had to work with existing non-generic code; to be checkable at compile-time; and with no change to the virtual machine. Included with Java 5 in 2004.

<http://homepages.inf.ed.ac.uk/wadler/gj>

Generics for .NET



Andrew Kennedy and Don Syme

Design and Implementation of Generics for the .NET Common Language Runtime

In *PLDI '01: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–12. ACM Press, 2001.

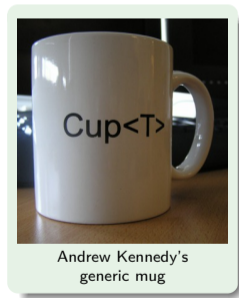
DOI: 10.1145/381694.378797

Work done over a similar period to Java, but with a different approach.

- Convinced Microsoft to modify the virtual machine
- Microsoft were prepared to make more significant language changes
- Had to support multiple-language working
- Included more flexible and powerful type features
- Provided more runtime support

Included with .NET Framework 2.0 in 2005.

https://is.gd/dng_hist



Algebraic Datatypes

Haskell

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```
Node (Leaf 3) (Leaf 4) : Tree int
```

OCaml

```
[1; 2; 3] : int list
```

```
# let rec sum list = match list with
```

```
| [] -> 0
```

```
| x::xs -> x + (sum xs);;
```

```
val sum : int list -> int = <fun>
```


Type Constructors

We have seen different examples of types that come in families:

- Generics in Java and C#/.NET;
- Algebraic datatypes in Haskell and OCaml.

These are all *parameterized* types: types that vary according to some *parameter*.

Each specific parameterized type is built 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

Where values and functions have types, types and constructors have *kinds*, e.g. `int : *` and `Tree : * -> *`

Parameterized Types in the Lambda Calculus

It is not hard to add specific parameterized types to the lambda calculus.

In fact we've already seen some with product and function space.

Examples of Parameterized Types

$(4, 5) : \text{num} \times \text{num}$ constructor '×' parameters **num** and **num**

$\lambda x. (x + 1) : \text{num} \rightarrow \text{num}$ constructor '→' parameters **num** and **num**

Adding further type constructors like **list** or **tree** is straightforward.

$$\frac{\text{Type } \tau}{\text{Type list } \tau}$$

$$\frac{\text{Type } \tau}{\text{Type tree } \tau}$$

All this has given us lots of types, but what about values of those types? What about functions that accept and return values of those types?

Outline

- 1 Opening
- 2 First-Class Functions
- 3 Parameterized Types
- 4 Polymorphism**
- 5 Closing

Polymorphism

Code is *polymorphic* when it can be used with values of different types.
One example is the use of virtual method calls in object-oriented code.

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

Each **Shape** *s* may actually be a **Square**, **Circle** or other implementation of **Shape**, each with its own implementation of `draw`.



These implementations may be entirely different, and possibly incompatible: consider **Picture.draw()** and **Cowboy.draw()**.



Flavours of Polymorphism

Ad-hoc Polymorphism

Classic object-oriented polymorphism: invoke method `a.draw()` and get whatever code is assigned to the target object `a` or its class.

Implementing this requires some attention to the *dispatch* of methods to determine the code finally executed.

Parametric Polymorphism

Operations that act similarly whatever the argument type: for example, sorting a list, or applying a function to every element of a collection.

Parametric polymorphism is heavily used in functional languages, and closely tied to parameterized types. In object-oriented languages it is usually known as *generic* programming.

Parametric Polymorphic Code

OCaml

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

```
length : 'a list -> int
```

```
# let rec map f = function
```

```
  | [] -> []
```

```
  | (x :: xs) -> (f x) :: (map f xs) ;;
```

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

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

Parametric Polymorphic Code

Java

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

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

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

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

    return result;
}
```

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

Outline

- 1 Opening
- 2 First-Class Functions
- 3 Parameterized Types
- 4 Polymorphism
- 5 Closing

Summary

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.

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*.

Polymorphism enables code to act on values of many types. With *ad-hoc polymorphism*, different values may be treated differently; with *parametric polymorphism* a single piece of code acts in the same way on many different argument types. This is one kind of *generic* programming, and parametric polymorphism in object-oriented languages is often known as *generics*.

Homework (1/2)

Read This



Luca Cardelli

https://is.gd/cardelli_types

Type Systems: Section 1 “Introduction”.

Chapter 97 of *The Computer Science and Engineering Handbook*, 2nd Edition

Watch This

https://is.gd/wadler_pat_video (Video, 43m)

Propositions as Types (Recorded at Strange Loop, September 2015)

Phil Wadler, Professor of Theoretical Computer Science, Edinburgh University



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` \leq `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?