Advances in Programming Languages

Parameterized Types and Polymorphism

Ian Stark

School of Informatics
The University of Edinburgh

Tuesday 27 September 2016
Semester 1 Week 2

THE UNIVERSITY *of* EDINBURGH

PLInG: Programming Language Interest Group

School of Informatics

Meeting: 1pm Thursday 29 September 2016 in IF 2.33

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 email to apl-students mailing list for more details on the PLInG mailing list and meetings.

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

Types appear widely in programming languages, 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.

Homework

Read This

Achim Jung
A Short Introduction to the Lambda Calculus http://is.gd/jung_lc

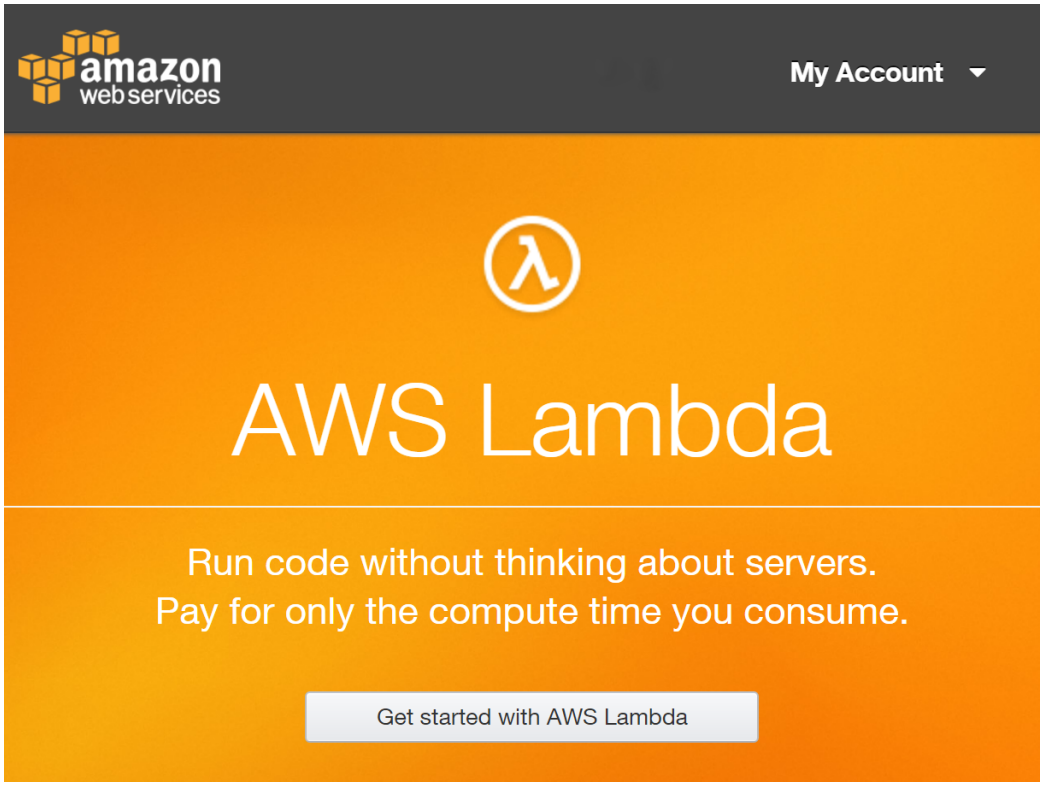
You may find it helpful to read pages 1–7 of Pierce's "Foundational Calculi for Programming Languages" alongside, although that's wholly untyped.
http://is.gd/pierce_fc

Extra Interest

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)

Amazon Web Services 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

Outline

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.

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

Andrew Kennedy's generic mug

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 : $* \rightarrow *$

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 '$\times$'	parameters num and num
$\lambda x.(x + 1) : \text{num} \to \text{num}$	constructor '$\to$'	parameters num and num

Adding further type constructors like list or tree is straightforward.

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

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

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)              // Use 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.

What is it Good For?

Are parameterized types useful?

Why not just use Object?

Is polymorphic code useful? For what?

Outline

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,...

$$\sigma \ ::= \ \tau \ | \ \forall\alpha.\sigma$$

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:

$$\mathsf{let}\, \mathrm{x} = \mathrm{M}\, \mathsf{in}\, \mathrm{N}$$

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

Summary

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. Specifically, *parametric polymorphism* gives a single piece of code that 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*.

Hindley-Milner types and *type inference* make it possible to write strongly-typed polymorphic code that is expressive but uncluttered by type annotations: with *algorithm W* or similar automatically identifying the *most general type* possible.

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?