Advances in Programming Languages
Terms and Types

Ian Stark

School of Informatics
The University of Edinburgh

Friday 23 September 2016
Semester 1 Week 1
From Lecture 1: What’s So Important About Language?

- **Abstraction:** Lift the level of operations you can describe

- **Programmability:** Build a new computer from the one you have

- **Expression:** Broaden your thoughts and the programs you can imagine

“To me programming is more than an important practical art. It is also a gigantic undertaking in the foundations of knowledge.”

David Sayre (one of the creators of FORTRAN) in conversation with Grace Hopper (one of the key advocates for COBOL), 1962
What’s in the course?

The lectures will cover four sample areas of “advances in programming languages”:

- **Types**: Parameterized, Polymorphic, Dependent, Refined
- Programming for Concurrency
- Augmented Languages for Correctness and Certification
- Programming for Memory Safety

Lectures also specify reading and exercises on the topics covered. This homework is not assessed, but it is essential in order to fully participate in the course.

There is substantial piece of written coursework which contributes 20% of your course grade. This requires investigation of a topic in programming languages and writing a 10-page report with example code.
Acknowledgements

This course is based on an original proposal by Stephen Gilmore. It has been developed over time by Ian Stark and David Aspinall, and continues to evolve from year to year. Things change: programming languages, the challenges that arise, and ways to meet them.

Ian Stark
Stephen Gilmore
David Aspinall
We might like a language that is:
- Easy to learn, quick to write, expressive, concise, powerful, supported, well-provided with libraries, cheap, popular, ...

It might help us to write programs that are:
- Readable, correct, fast, reliable, predictable, maintainable, secure, robust, portable, testable, verifiable, composable, ...

It might help us address challenges in:
- Multicore architectures, distributed computing, warehouse-scale computation, programming the web, quantum computing, ...

Course: Advances in Programming Languages
Lecturer: Ian Stark
Level: Undergraduate year 4, year 5 and MSc students (10 credit points at Level 11)
When: 1510–1600 Tuesday & Friday
Where: George Square Lecture Theatre (or possibly elsewhere...)
Web: https://blog.inf.ed.ac.uk/apl16

Brainstem
- Midbrain
-pons
-medulla
-cerebellum
-vestibulocerebellum
-spinocerebellum
-cerebrocerebellum
-frontal lobe
-parietal lobe
-temporal lobe
-occipital lobe
-cerebrum
This first block of lectures in this course looks at some uses of *types*.

- **Terms and Types**
- **Parameterized Types and Polymorphism**
- **Higher Polymorphism**
- **Dependent Types**

The study of *Type Theory* is a 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. Types
2. Lambda Calculus
3. First-Class Functions in Programming Languages
4. Closing
Outline

1. Types
2. Lambda Calculus
3. First-Class Functions in Programming Languages
4. Closing
Homework

1. Read the Wikipedia article on *History of programming languages*. (If you find it’s missing something, fix that.)

2. Pick a programming language you don’t already know, and find out the following.

   - Does it assign types to distinguish between things like numbers, strings, or functions?
   - Does it check these are used correctly?
   - How does it do that? When does it do that?

Bring your answers along to the lecture.
Some types

A selection of types from some languages.

C/C++

int, long, float, unsigned int, char
int [], char*, char&, int(*)(float,char)
extern const volatile unsigned long int

OCaml

int, int64, bool, char, string, unit
string*string, int list, bool array
int->int, int->string->char, 'a list -> 'a list

Java

Object, byte[], boolean
StringBuffer, LinkedList, TreeSet, ArrayList<String>
IllegalPathStateException, BeanContextServiceRevokedListener
What do people do with types?

- Type checking
- Static type checking
- Dynamic type checking
- Type annotation
- Type inference
- Structural typing
- Nominative typing
- Duck typing

- Subtyping
- Effect types
- Session types
- Refinement types
- Soft typing
- Gradual typing
- Dynamic types
- Blame typing
To find out more...

...and lots more

Outline

1. Types
2. Lambda Calculus
3. First-Class Functions in Programming Languages
4. Closing
The Lambda Calculus or $\lambda$-calculus is a formal system for modelling and reasoning about computation. Its origins lie in mathematics and logic, and it was created to give a structure for working with the foundations of logic on a par with more familiar mathematical constructions like groups and vector spaces.

We may draw the analogy of a three dimensional geometry used in describing physical space, a case for which, we believe, the presence of such a situation is more commonly recognized.

Alonzo Church, Princeton, 1931
We define a set Term of terms using the following rules, where \( \text{Var} \) is some set of variables \( x, y, \ldots \)

### Rules for Constructing Lambda-Calculus Terms

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Var} \ x )</td>
<td>Variable</td>
</tr>
<tr>
<td>( \text{Term} \ x )</td>
<td>( \text{Var} \ x ) here means “( x ) is a variable”</td>
</tr>
<tr>
<td>( \text{Var} \ x ) ( \text{Term} \ M )</td>
<td>Function abstraction</td>
</tr>
<tr>
<td>( \text{Term} \ \lambda x. M )</td>
<td>( \text{Term} \ M ) here means “( M ) is a term”</td>
</tr>
<tr>
<td>( \text{Term} \ M_1 \text{ Term} \ M_2 )</td>
<td>Function application</td>
</tr>
<tr>
<td>( \text{Term} \ M_1 M_2 )</td>
<td></td>
</tr>
</tbody>
</table>

Each rule states that when we have all the things above the line (the hypotheses) then we can deduce the thing below the line (the conclusion).

Taken together, these rules describe Term as an *inductively defined set*, the smallest set closed under all the rules.
Terms

We define a set \textbf{Term} of terms using the following rules, where \textbf{Var} is some set of variables $x, y, \ldots$

\textbf{Rules for Constructing Lambda-Calculus Terms}

\begin{align*}
\text{Var } x & \quad \text{Variable} \\
\frac{\text{Term } x}{\text{Term } x} & \\
\text{Var } x & \text{ here means “} x \text{ is a variable”} \\
\text{Term } M & \text{ here means “} M \text{ is a term”} \\
\frac{\text{Var } x \quad \text{Term } M}{\text{Term } \lambda x. M} & \quad \text{Function abstraction} \\
\frac{\text{Term } M_1 \quad \text{Term } M_2}{\text{Term } M_1 M_2} & \quad \text{Function application}
\end{align*}

In writing terms we use parentheses where necessary to disambiguate the structure. Application is left-associative, so $FMN$ means $(FM)N$.

To help with examples we might also include as terms some set \textbf{Const} of constants such as $2, +, \text{sqrt}, \ldots$
Bound and Free Variables

Variables in a term that match some enclosing $\lambda$ are *bound* by that $\lambda$.
All other variables mentioned in a term are *free*.

**Examples**

- $\lambda n. (n + 1)$  
  Variable $n$ is bound

- $\lambda x. (\lambda y. (x * y * z))$  
  Here $x$ and $y$ are bound and $z$ is free

- $(\lambda f. f(p + q))(\text{sqrt})$  
  Here $f$ is bound while $p$ and $q$ are free
Substitution

**Alpha Equivalence**

We say that terms like \( \lambda x.(x + 1) \) and \( \lambda y.(y + 1) \) are \( \alpha \)-equivalent, and usually consider them to represent the same lambda-term.

Replacing one variable with another like this is called \( \alpha \)-conversion.

**Capture-Avoiding Substitution**

We write \( M\{N/x\} \) to represent the term \( M \) with every occurrence of variable \( x \) replaced with the term \( N \).

If \( N \) contains a variable \( y \) that is bound in \( M \), there is a risk of it being captured by the binding. Usually this is a bad thing, and we should \( \alpha \)-convert the binding in \( M \) first to give capture-avoiding substitution.

**Example**

\[
(\lambda x.(x*y)) \{(x + x)/y\} = (\lambda z.(z*y)) \{(x + x)/y\} = (\lambda z.(z*(x+x)))
\]
Reduction

The $\beta$-reduction rule is central to the role of lambda-abstractions as functions, and to the lambda-calculus as a model of computation.

\[ (\lambda x. M) N \rightarrow M\{N/x\} \]

There is much more to lambda-calculus reduction — rules for constants, applying $\beta$ within terms, simultaneous $\beta$-reduction, confluence, normalization — but for now it’s enough to see that the $\beta$ rule captures the effect of function application.
Types for Terms

So far we have had an *untyped* system: rules for building up terms, \( \alpha \)-equivalence, \( \beta \)-reduction all work by rearranging symbols. This works, and the *untyped lambda-calculus* is a complete computational framework. (Look up the “Church-Turing thesis”)

The *typed* lambda-calculus constrains the system a little, by specifying what sort of arguments a function will accept and what sort of result it returns. This is particularly appropriate when we have constants with intrinsic types.

We write \( M : \tau \) to indicate that term \( M \) has type \( \tau \).

**Examples**

\[
\begin{align*}
sqrt 25 & : \text{num} & \lambda a. (\lambda b. (a + b)) & : \text{num} \rightarrow (\text{num} \rightarrow \text{num}) \\
\lambda x. (x + 2) & : \text{num} \rightarrow \text{num} & \lambda f. (f(4) \ast x) & : (\text{num} \rightarrow \text{num}) \rightarrow \text{num}
\end{align*}
\]
Simply-Typed Lambda Calculus

A type context $\Gamma$ is a set of variables with types.

$$\Gamma = \{x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n\}$$

A type declaration $\Gamma \vdash M : \tau$ asserts that if the variables in $\Gamma$ have the types listed, then term $M$ has type $\tau$.

Rules for Constructing Typed Lambda-Calculus Terms

- **Variable**
  $$\frac{}{\Gamma \vdash x : \tau} \quad \text{if } x : \tau \in \Gamma$$

- **Function abstraction**
  $$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 . M) : \tau_1 \rightarrow \tau_2}$$

- **Function application**
  $$\frac{\Gamma \vdash F : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash M : \tau_1}{\Gamma \vdash FM : \tau_2}$$
Styles of Typing

The rules just given require lambda-abstraction to include the type of the bound variable:

$$(\lambda x: \tau. x) : \tau \rightarrow \tau$$

This is called *Church-style* typing.

Earlier, we saw some terms without these internal type statements:

$$\lambda a. (\lambda b. (a + b)) : \text{num} \rightarrow (\text{num} \rightarrow \text{num})$$

This is *Curry-style* typing.

Both styles are viable, with a range of slight variations in common use.

In fact, this applies more broadly for formal systems like this: while most presentations are internally consistent about syntax, rules, and terminology there may be many minor variations between different presentations. For example, writing typed variables as $x_\tau$ rather than $x:\tau$. 
Why λ?

Apparently Church originally used a “hat” over the bound variable

\[ \hat{x}.(x + 1) \]
Why $\lambda$?

Apparently Church originally used a “hat” over the bound variable

$$\hat{x}.(x + 1)$$

which for a more linear notation was moved to the left by printers and enlarged

$$\land x.(x + 1)$$
Why λ?

Apparently Church originally used a “hat” over the bound variable

\( \hat{x}.(x + 1) \)

which for a more linear notation was moved to the left by printers and enlarged

\( \wedge x.(x + 1) \)

to give something very like the uppercase Greek lambda

\( \Lambda x.(x + 1) \)
Why $\lambda$?

Apparently Church originally used a “hat” over the bound variable

$$\hat{x}. (x + 1)$$

which for a more linear notation was moved to the left by printers and enlarged

$$\land x. (x + 1)$$

to give something very like the uppercase Greek lambda

$$\Lambda x. (x + 1)$$

and to avoid confusion with the letter “A” this was replaced with a lowercase lambda

$$\lambda x. (x + 1) .$$

So the story goes.
Pairing and Tuples

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>Term (M_1) Term (M_2)</td>
<td>(\Gamma \vdash M_1 : \tau_1) (\Gamma \vdash M_2 : \tau_2)</td>
<td>(\text{fst}(M_1, M_2) \rightarrow M_1)</td>
</tr>
<tr>
<td>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>Term (\text{fst})</td>
<td>(\vdash \text{fst} : (\tau_1 \times \tau_2) \rightarrow \tau_1)</td>
<td></td>
</tr>
<tr>
<td>Term (\text{snd})</td>
<td>(\vdash \text{snd} : (\tau_1 \times \tau_2) \rightarrow \tau_2)</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
Curried Functions

Lambda-calculus functions taking multiple arguments can be written either using tuples or a function that returns a function.

Examples

\[ \lambda p: (\text{num} \times \text{num}) . (\text{fst} \ p + \text{snd} \ p) \ : \ (\text{num} \times \text{num}) \rightarrow \text{num} \]

\[ \lambda a: \text{num}.(\lambda b: \text{num}.(a + b)) \ : \ \text{num} \rightarrow (\text{num} \rightarrow \text{num}) \]

Passing from \((\text{num} \times \text{num}) \rightarrow \text{num}\) to \(\text{num} \rightarrow (\text{num} \rightarrow \text{num})\) is called **Currying**, and the latter type is usually written as \(\text{num} \rightarrow \text{num} \rightarrow \text{num}\).
Higher-Order Functions

A term has *ground type* or is *zero-order* if it is not a function.

A function is *first-order* if it only takes arguments that are of ground type.

A function is *second-order* if it takes a first-order function as an argument.

A function is *order* \( n \) if it takes an order \((n - 1)\) function as an argument.

All functions of second order and above are *higher-order* functions.

5 : num  true : bool  \( \sqrt{3} \) : real

negate : num \( \rightarrow \) num

xor : bool \( \rightarrow \) bool \( \rightarrow \) bool

power : num \( \rightarrow \) (real \( \rightarrow \) real)

integrate : (real \( \rightarrow \) real) \( \rightarrow \) (real \( \rightarrow \) real)

is-zero-at : num \( \rightarrow \) (num \( \rightarrow \) num) \( \rightarrow \) bool

apply : (\( \sigma \rightarrow \tau \)) \( \rightarrow \) \( \sigma \rightarrow \tau \)

\( \ldots \)
There are many features of typed and untyped lambda calculus that are important and interesting, but will not appear in this course.

- Confluence of reduction rules
- Evaluation strategies: call-by-value; call-by-name; parallel; optimal
- Recursive functions
- Type inference
- ...

There are also many things that can be encoded in the lambda-calculus, or added to it, or sometimes both, that we shall not look at further.

- Natural numbers
- Booleans, sets, data structures
- Objects
- Stateful computation, input/output, side-effects
- ...
Outline

1. Types
2. Lambda Calculus
3. First-Class Functions in Programming Languages
4. Closing
First-Class Functions

The lambda-calculus originated as a 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.

\[
\text{(lambda (x y) (+ x y)) ; Add two numbers}
\]

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

\[
\text{(fn p => 2*p) } (* \text{ Double a value, Standard ML syntax } *)
\]
as well as Haskell:

\[
\text{\textbackslash p \rightarrow \textbackslash q \rightarrow p ++ q } \text{ -- Concatenate two lists.}
\]
Examples

Java 8

q -> q+1

(a,b) -> Math.sqrt(a*a + b*b)

(String s, String t) -> { String result = s + t; return result; }

Smalltalk

[ :x | x*x*x*x ]

Scala

(f: String=>Int, s: String) => f(s)
Closures

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

\[(\lambda x:\text{num}. (\lambda y:\text{num}. (x \times y)) \ 5 \rightarrow \lambda y:\text{num}. (5 \times y)\]

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

\[\{x = 5; (\lambda y:\text{num}. x \times y)\}\]

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 8, for example, disallows closures that refer to mutable state.
import java.util.function.Predicate;

public class Checker {

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

}
public void runLengthTests (Checker c) {
  int n = 0;

  Predicate<String> lengthTest =
    (String y) -> {
      System.out.println("Comparing to: "+n);  // error
      return (y.length() >= n);           // error
    };

  n = 4;  c.runCheck(lengthTest);
  n = 8;  c.runCheck(lengthTest);
}

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

1 Types

2 Lambda Calculus

3 First-Class Functions in Programming Languages

4 Closing
Summary

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.
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)