

# Advances in Programming Languages

## Terms and Types

Ian Stark

School of Informatics  
The University of Edinburgh

Thursday 20 September 2018  
Semester 1 Week 1



# Summary

- **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
- [Your choice here: Memory Safety and Rust, or . . . , or . . . ]

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

## Regular Substantive Slide

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, ...

Ian Stark

APL /

September 20, 2018

## Announcement Slide

!

**Course:** Advances in Programming Languages

**Lecturer:** Ian Stark

**Level:** Undergraduate Year 4, Year 5 and MSc students (10 credit points at Level 11)

**When:** 1610–1700 Monday & Thursday

**Where:** LG.11 David Hume Tower / Gaddum Lecture Theatre

**Web:** <https://wp.inf.ed.ac.uk/apl18>

**Learn:** Advances in Programming Languages (2018-2019)[SV1-SEM1]

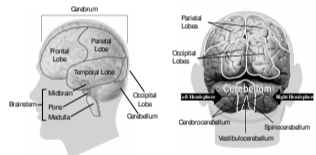
Ian Stark

APL /

September 20, 2018

## Bonus Off-Syllabus Slide

+



Ian Stark

APL /

September 20, 2018

# Topic: Some Types in Programming Languages

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

# 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

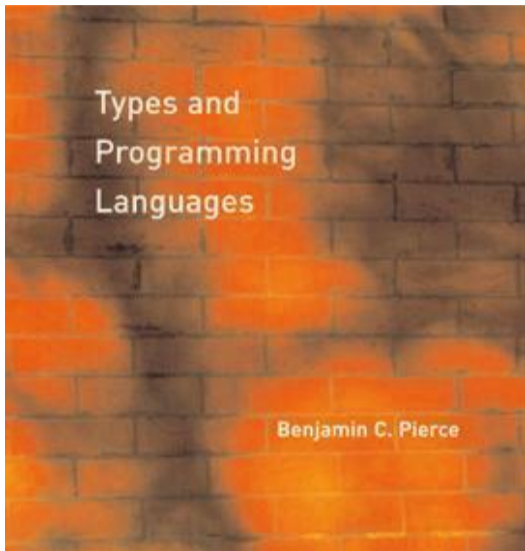
**int, int64, bool, char, string, unit**  
**string\*string, int list, bool array**  
**int->int, int->string->char, 'a list -> 'a list**

## Java

**Object, byte[], boolean**  
**StringBuffer, LinkedList, TreeSet, ArrayList<String>**  
**IllegalPathStateException, BeanContextServiceRevokedListener**

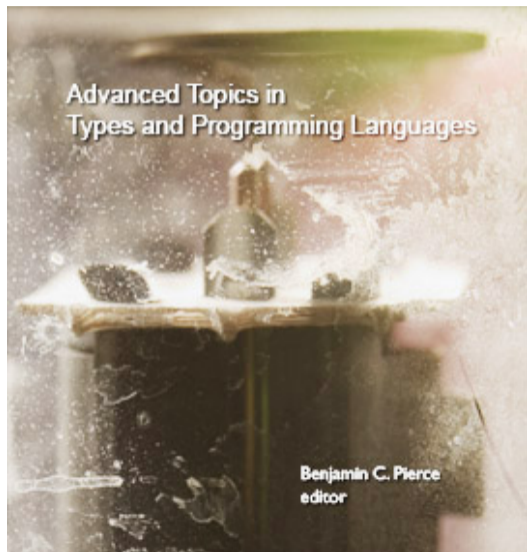
# What do people do with types?

- 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



Benjamin C. Pierce.

*Types and Programming Languages.*  
MIT Press, 2002.



Benjamin C. Pierce, editor.  
*Advanced Topics in Types and  
Programming Languages.*  
MIT Press, 2005.

# Lambda Calculus

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

## A SET OF POSTULATES FOR THE FOUNDATION OF LOGIC.<sup>1</sup>

BY ALONZO CHURCH.<sup>2</sup>

1. Introduction. In this paper we present a set of postulates for the foundation of formal logic, in which we avoid use of the free, or real, variable, and in which we introduce a certain restriction on the law of excluded middle as a means of avoiding the paradoxes connected with the mathematics of the transfinite.

Our reason for avoiding use of the free variable is that we require that every combination of symbols belonging to our system, if it represents a proposition at all, shall represent a particular proposition, unambiguously, and without the addition of verbal explanations. That the use of the free variable involves violation of this requirement, we believe is readily seen. For example, the identity

$$(1) \quad a(b+c) = ab+ac$$

in which  $a$ ,  $b$ , and  $c$  are used as free variables, does not state a definite proposition unless it is known what values may be taken on by these variables, and this information, if not implied in the context, must be given by a verbal addition. The range allowed to the variables  $a$ ,  $b$ , and  $c$  might consist of all real numbers, or of all complex numbers, or of some other set, or the ranges allowed to the variables might differ, and for each possibility equation (1) has a different meaning. Clearly, when this equation is written alone, the proposition intended has not been completely translated into symbolic language, and, in order to make the translation complete, the necessary verbal addition must be expressed by means of the symbols of formal logic and included, with the equation, in the formula used to represent the proposition. When this is done we obtain, say,

$$(2) \quad R(a) R(b) R(c) \supset_{\text{alt}} a(b+c) = ab+ac$$

where  $R(x)$  has the meaning, " $x$  is a real number," and the symbol  $\supset_{\text{alt}}$  has the meaning described in §§ 5 and 6 below. And in this expression there are no free variables.

<sup>1</sup> Received October 5, 1931.

<sup>2</sup> This paper contains, in revised form, the work of the author while a National Research Fellow in 1928-29.

# Terms

We define a set **Term** of terms using the following rules, where **Var** is some set of variables  $x, y, \dots$

## Rules for Constructing Lambda-Calculus Terms

$$\frac{\text{Var } x}{\text{Term } x} \quad \text{Variable}$$

$$\frac{\text{Var } x \quad \text{Term } M}{\text{Term } \lambda x.M} \quad \text{Function abstraction}$$

**Var**  $x$  here means “ $x$  is a variable”  
**Term**  $M$  here means “ $M$  is a term”

$$\frac{\text{Term } M_1 \quad \text{Term } M_2}{\text{Term } M_1 M_2} \quad \text{Function application}$$

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 **Term** of terms using the following rules, where **Var** is some set of variables  $x, y, \dots$

## Rules for Constructing Lambda-Calculus Terms

$$\frac{\text{Var } x}{\text{Term } x} \quad \text{Variable}$$

$$\frac{\text{Var } x \quad \text{Term } M}{\text{Term } \lambda x.M} \quad \text{Function abstraction}$$

**Var**  $x$  here means “ $x$  is a variable”  
**Term**  $M$  here means “ $M$  is a term”

$$\frac{\text{Term } M_1 \quad \text{Term } M_2}{\text{Term } M_1 M_2} \quad \text{Function application}$$

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 **Const** of constants such as  $2, +, \text{sqrt}, \dots$

# 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.

## Beta-Reduction

$$(\lambda x.M) N \longrightarrow M\{N/x\}$$

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

$\text{sqrt } 25 : \text{num}$

$\lambda x. (x + 2) : \text{num} \rightarrow \text{num}$

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

$\lambda f. (f(4) * x) : (\text{num} \rightarrow \text{num}) \rightarrow \text{num}$

# Simply-Typed Lambda Calculus

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

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

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

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

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

$$\frac{\Gamma \vdash F : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash M : \tau_1}{\Gamma \vdash FM : \tau_2} \quad \text{Function application}$$

# 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 small variations between different presentations. For example, writing typed variables as  $x_\tau$  rather than  $x:\tau$ .



Alonzo Church (1903–1995)



Haskell Curry (1900–1982)

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

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

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*  $(\tau_1 \times \tau_2)$ .

## Terms

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

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

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

## Typing

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

$$\frac{}{\vdash \text{fst} : (\tau_1 \times \tau_2) \rightarrow \tau_1}$$

$$\frac{}{\vdash \text{snd} : (\tau_1 \times \tau_2) \rightarrow \tau_2}$$

## Reduction

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

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

This can be extended to *tuples* of arbitrary size  $(M_1, M_2, \dots, M_n)$ , 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

$$\lambda p:(\text{num} \times \text{num}) . (\text{fst } p + \text{snd } p) \quad : \quad (\text{num} \times \text{num}) \rightarrow \text{num}$$
$$\lambda a:\text{num} . (\lambda b:\text{num} . (a + b)) \quad : \quad \text{num} \rightarrow (\text{num} \rightarrow \text{num})$$

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.

5 : num   true : bool   sqrt(3) : real

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

negate : num  $\rightarrow$  num  
xor : bool  $\rightarrow$  bool  $\rightarrow$  bool  
power : num  $\rightarrow$  (real  $\rightarrow$  real)

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

integrate : (real  $\rightarrow$  real)  $\rightarrow$  (real  $\rightarrow$  real)  
is-zero-at : num  $\rightarrow$  (num  $\rightarrow$  num)  $\rightarrow$  bool  
apply : ( $\sigma \rightarrow \tau$ )  $\rightarrow$   $\sigma \rightarrow \tau$

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.

# Not Today

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
- ...

# 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.

# 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.

“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](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](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](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)