IA014: Advanced Functional Programming

Course information
1. History of λ-calculus

Jan Obdržálek obdrzalek@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Course information

Course organization

Lectures

- **Thursday, 12 noon, D2**
- language: English
- slides will be available (IS)

Exam

- midterm exam (worth 15 % of the points)
- final exam (85 %)
- both written
- in English, but you may answer in Czech
- *k* and *z* completion possible

Prerequisites

- no *formal* requirements
- some experience with functional programming
  - evaluation, recursion, abstract data types, pattern matching, lists (including comprehensions), higher-order functions, . . .
  - e.g. to the extent covered by

    IB015 – Non-imperative programming
  - you should be able to write simple functional programs
  - you will have first few weeks to catch-up on FP, if you are willing to work
- we will use mostly *Haskell*
  - but if you know ML (OCaml, F#, . . .) you should be fine (see the recommended literature)

Topics covered

- history of λ-calculus (and functional programming)
- λ-calculus
  - untyped λ-calculus
  - simply typed λ-calculus
  - polymorphic λ-calculus (system F)
- type inference (Hindley-Milner alg.)
- type system extensions
- type classes
- functors, monads
- monad transformers
- purely functional data structures
- concurrency
- applications
  - DSL - Domain specific languages
  - Quickcheck - type based property testing
  - dependent types: Agda and Coq

Reading

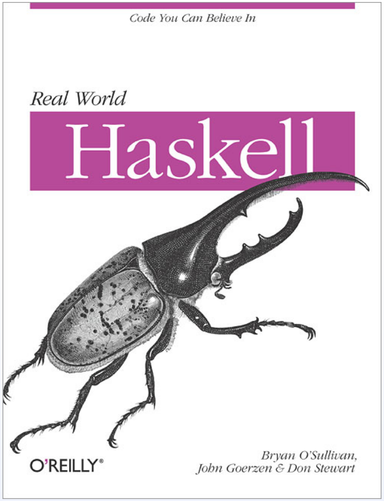
- There is no book covering the whole course.
- Some topics cannot be found in any book at all.
- I will use *various research papers* (available in IS)

`https://is.muni.cz/auth/el/1433/podzim2014/IA014/index.qwarp`

some (more or less) general books

- B. O'Sullivan, J. Goerzen and D. Stewart: *Real World Haskell*
- M. Lipovača: *Learn You a Haskell for Great Good!*
- G. Michaelson: *An Introduction to Functional Programming Through Lambda Calculus*
- C. Okasaki: *Purely Functional Data Sructures*
- B. Pierce: *Types and Programming Languages*

Real World Haskell



Code You Can Believe In

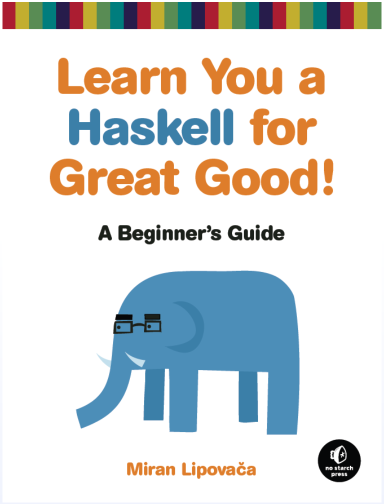
Real World Haskell

O'REILLY®

Bryan O'Sullivan, John Goerzen & Don Stewart

`http://book.realworldhaskell.org/`

Learn You a Haskell


Learn You a Haskell for Great Good!

A Beginner's Guide

Miran Lipovača

no starch press

http://learnyouahaskell.com/

(Short) History of λ-calculus (and functional programming)

Based on a talk “Church’s Coincidences” by Phil Wadler.

David Hilbert (1862-1943)

Entscheidungsproblem

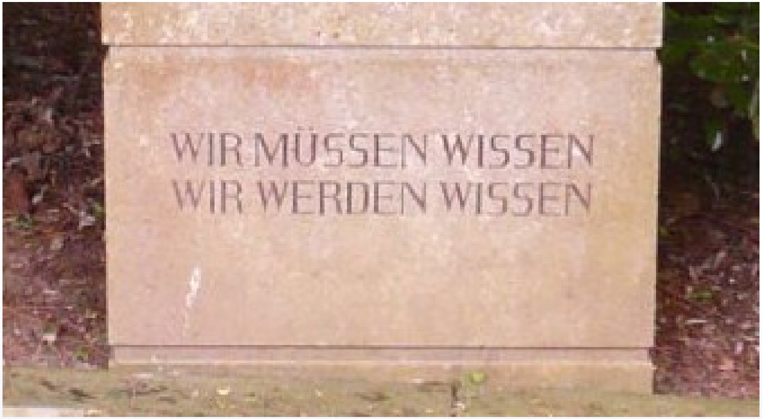
Hilbert, 1928

Does there exists an algorithm with the following properties:
INPUT: formula ϕ of First Order logic
(+ finite number of extra axioms, e.g. Peano arithmetic)
OUTPUT: YES iff ϕ is true (universally valid)

- Many problems could then be *automatically* solved:
  - Goldbach Conjecture
  - Riemann Hypothesis
  - Diophantine Equations (Hilbert's 10th problem)
  - …
- "Little detail": what is meant by *algorithm*?
- We need a notion of *effective computability*

David Hilbert (1862-1943)

David Hilbert (1862-1943)



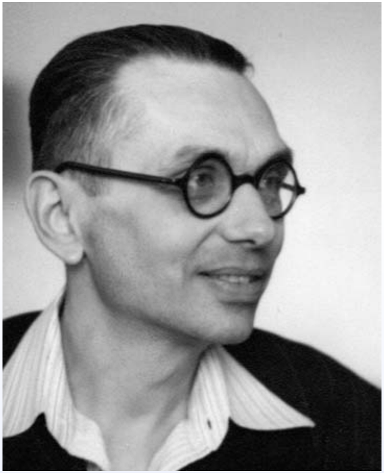
David Hilbert (1862-1943)

For the mathematician there is no Ignorabimus, and, in my opinion, not at all for natural science either. . . . The true reason why [no one] has succeeded in finding an unsolvable problem is, in my opinion, that there is no unsolvable problem. In contrast to the foolish Ignorabimus, our credo avers: We must know, We shall know.

D. Hilbert

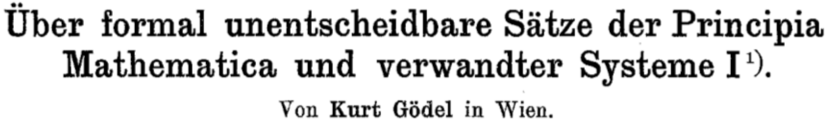
Königsberg, 8 September 1930
Society of German Scientists and Physicians

Kurt Gödel (1906-1978)



Kurt Gödel (1906-1978)

Gödel's Incompleteness Theorem



Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I[1].

Von **Kurt Gödel** in Wien.

On Formally Undecidable Propositions of Principia Mathematica and Related Systems I (1931)

For any consistent, effectively generated formal theory that proves certain basic arithmetic truths, there is an arithmetical statement that is true, but not provable in the theory.

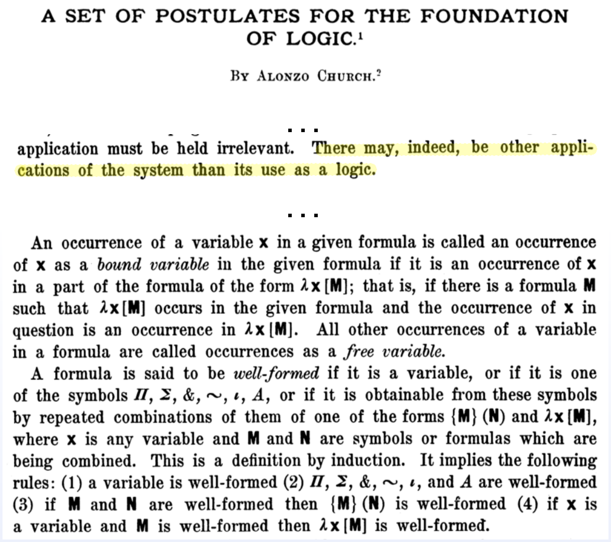
Königsberg, 7 September 1930

Conference on Epistemology

held jointly with the meetings of Society of German Scientists and Physicians

Alonzo Church (1903-1995)

A. Church – λ-calculus (1932)



A SET OF POSTULATES FOR THE FOUNDATION OF LOGIC.[1]

BY ALONZO CHURCH.[2]

. . .

application must be held irrelevant. There may, indeed, be other applications of the system than its use as a logic.

. . .

An occurrence of a variable **x** in a given formula is called an occurrence of **x** as a *bound variable* in the given formula if it is an occurrence of **x** in a part of the formula of the form $\lambda \mathbf{x}[\mathbf{M}]$; that is, if there is a formula **M** such that $\lambda \mathbf{x}[\mathbf{M}]$ occurs in the given formula and the occurrence of **x** in question is an occurrence in $\lambda \mathbf{x}[\mathbf{M}]$. All other occurrences of a variable in a formula are called occurrences as a *free variable*.

A formula is said to be *well-formed* if it is a variable, or if it is one of the symbols Π, Σ, &, $\sim$, ι, A, or if it is obtainable from these symbols by repeated combinations of them of one of the forms $\{\mathbf{M}\}(\mathbf{N})$ and $\lambda \mathbf{x}[\mathbf{M}]$, where **x** is any variable and **M** and **N** are symbols or formulas which are being combined. This is a definition by induction. It implies the following rules: (1) a variable is well-formed (2) Π, Σ, &, $\sim$, ι, and A are well-formed (3) if **M** and **N** are well-formed then $\{\mathbf{M}\}(\mathbf{N})$ is well-formed (4) if **x** is a variable and **M** is well-formed then $\lambda \mathbf{x}[\mathbf{M}]$ is well-formed.

λ-calculus (1932)

- formal system of *mathematical logic*
- based on function *abstraction* and *application* using *variable binding* and *substitution*
- intended to be a *foundation of mathematics*

$$f(x) = x^2 + x + 42$$
$$\Downarrow$$
$$f \equiv \lambda x.x^2 + x + 42$$

Notation

Then	Now	
x	x	variable
$\lambda\mathrm{x}[\mathrm{N}]$	$(\lambda\mathrm{x}.\mathrm{N})$	abstraction
$\{\mathrm{L}\}(\mathrm{M})$	$(\mathrm{L\ M})$	application

λ-definability

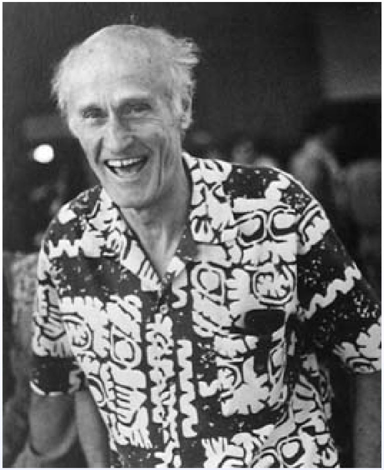
By 1932, the following could be modelled by λ-calculus:

- natural numbers
- +1 (successor)
- addition, multiplication
- exponentiation
- … *pretty much everything!*

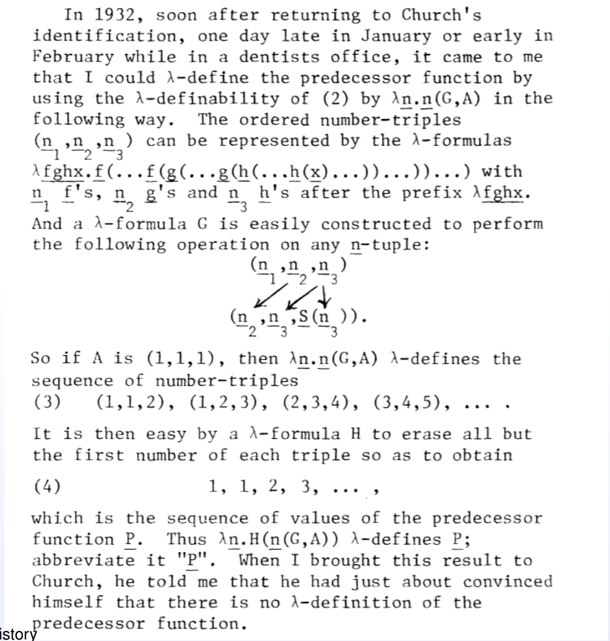
Problem!

- How to define the predecessor (-1) operation?
- If that would not be definable, then λ-calculus does not capture the notion of effective computability!
- Solution: S. Kleene, 1932

Stephen Kleene (1909-1994)



S. Kleene – predecessor (1932)



In 1932, soon after returning to Church's identification, one day late in January or early in February while in a dentists office, it came to me that I could λ-define the predecessor function by using the λ-definability of (2) by $\lambda \underline{n}.\underline{n}(G,A)$ in the following way. The ordered number-triples $(\underline{n}_1, \underline{n}_2, \underline{n}_3)$ can be represented by the λ-formulas $\lambda \underline{fghx}.\underline{f}(\ldots\underline{f}(\underline{g}(\ldots\underline{g}(\underline{h}(\ldots\underline{h}(\underline{x})\ldots))\ldots))\ldots)$ with $\underline{n}_1$ $\underline{f}$'s, $\underline{n}_2$ $\underline{g}$'s and $\underline{n}_3$ $\underline{h}$'s after the prefix $\lambda\underline{fghx}$. And a λ-formula G is easily constructed to perform the following operation on any $\underline{n}$-tuple:

$$\begin{tikzcd}(\underline{n}_1, & \underline{n}_2, \arrow[dl] & \underline{n}_3) \arrow[dl] \arrow[d] \\ (\underline{n}_2, & \underline{n}_3, & \underline{S}(\underline{n}_3)).\end{tikzcd}$$

So if A is (1,1,1), then $\lambda \underline{n}.\underline{n}(G,A)$ λ-defines the sequence of number-triples

(3) (1,1,2), (1,2,3), (2,3,4), (3,4,5), ...

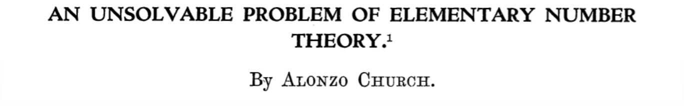
It is then easy by a λ-formula H to erase all but the first number of each triple so as to obtain

(4) 1, 1, 2, 3, ... ,

which is the sequence of values of the predecessor function $\underline{P}$. Thus $\lambda\underline{n}.H(\underline{n}(G,A))$ λ-defines $\underline{P}$; abbreviate it "$\underline{P}$". When I brought this result to Church, he told me that he had just about convinced himself that there is no λ-definition of the predecessor function.

λ-calculus – Undecidability (1936)

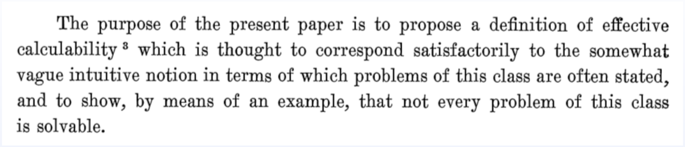
1935 – Kleene and Rosser showed λ-calculus to be *inconsistent*
1936 – Church publishes the computational part *(numerals etc.)*



AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.[1]

By Alonzo Church.

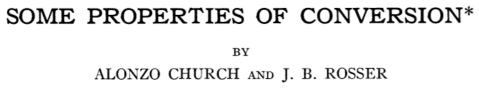
. . .



The purpose of the present paper is to propose a definition of effective calculability[3] which is thought to correspond satisfactorily to the somewhat vague intuitive notion in terms of which problems of this class are often stated, and to show, by means of an example, that not every problem of this class is solvable.

First undecidable problem: equivalence of two λ-terms

λ-calculus – Consistency (1936)

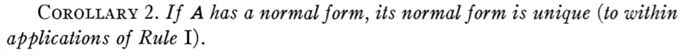


SOME PROPERTIES OF CONVERSION*

BY

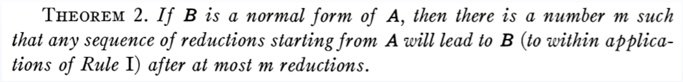
ALONZO CHURCH AND J. B. ROSSER

. . .



Corollary 2. *If* **A** *has a normal form, its normal form is unique (to within applications of Rule* I).

. . .



Theorem 2. *If* **B** *is a normal form of* **A**, *then there is a number m such that any sequence of reductions starting from* **A** *will lead to* **B** (*to within applications of Rule* I) *after at most m reductions.*

Proof that β-reduction is confluent.

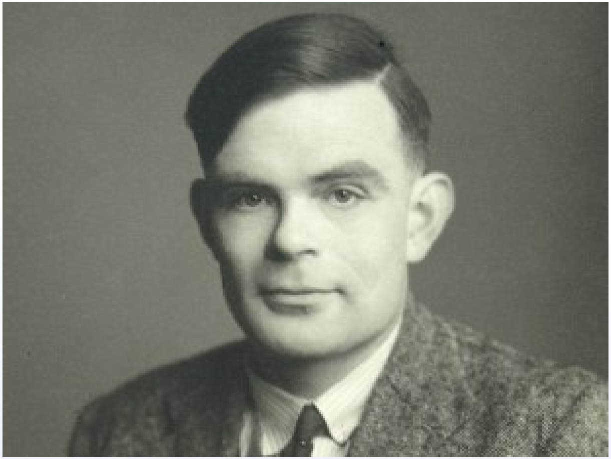
Effective Computability Models

- Alonzo Church: Lambda calculus
  An unsolvable problem of elementary number theory (Abstract)
  Bulletin the American Mathematical Society, May 1935

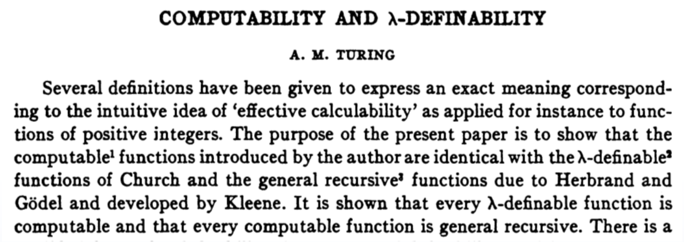
Two other notions defined independently:

- Stephen C. Kleene: Recursive functions
  General recursive functions of natural numbers (Abstract)
  Bulletin the American Mathematical Society, July 1935
- Alan M. Turing: Turing machines
  On computable numbers, with an application to the Entscheidungsproblem
  Proceedings of the London Mathematical Society, received 25 May 1936

Alan Turing (1912-1954)



A.Turing – Equivalence (1937)

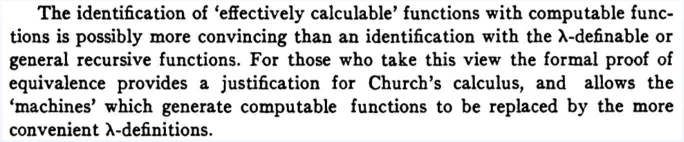


COMPUTABILITY AND λ-DEFINABILITY

A. M. TURING

Several definitions have been given to express an exact meaning corresponding to the intuitive idea of 'effective calculability' as applied for instance to functions of positive integers. The purpose of the present paper is to show that the computable[1] functions introduced by the author are identical with the λ-definable[2] functions of Church and the general recursive[3] functions due to Herbrand and Gödel and developed by Kleene. It is shown that every λ-definable function is computable and that every computable function is general recursive. There is a

. . .



The identification of 'effectively calculable' functions with computable functions is possibly more convincing than an identification with the λ-definable or general recursive functions. For those who take this view the formal proof of equivalence provides a justification for Church's calculus, and allows the 'machines' which generate computable functions to be replaced by the more convenient λ-definitions.

Typed λ-calculi ($\lambda \rightarrow$)

Two flavors

- Implicitly typed
  - Haskell B. Curry, 1934
  - $\mathrm{I} = (\lambda \mathrm{x.x}) : A \rightarrow A$
  - $\mathrm{I} = (\lambda \mathrm{x.x}) : (A \rightarrow B) \rightarrow (A \rightarrow B)$
- Explicitly typed
  - Alonzo Church, 1940
  - $\mathrm{I}_A = (\lambda \mathrm{x{:}A.x}) : A \rightarrow A$
  - $\mathrm{I}_{A \rightarrow B} = (\lambda \mathrm{x{:}(A \rightarrow B).x}) : (A \rightarrow B) \rightarrow (A \rightarrow B)$

Later developments

- Higher-order λ-calculi
  - Girard, 1972
  - *system F*, *system Fω*

Models for λ-calculus

Is there is *set theoretic model* for λ-calculus?

Naturally, we would need a set D isomorphic to the function space $D \to D$.

Problem: D and $D \to D$ have different cardinality!

Solution: D. Scott 1969

- model $\mathcal{D}_\infty$
- consider only *continuous functions* with appropriate topology

  model in cartesian closed category of complete lattices and Scott continuous functions
- then such domain D can be found

Led to the development of denotational semantics.

Functional programming timeline

- Lisp (McCarthy, 1960)
- Iswim (Landin, 1966)
- Scheme (Steele and Sussman, 1975)
- ML (Milner, Gordon, Wadsworth, 1979)
- Miranda (Turner, 1985)
- Haskell (Hudak, Peyton Jones, Wadler, 1987)
- OCaml (Leroy, 1996)
- Erlang (Armstrong, Virding, Williams, 1996)
- Scala (Odersky, 2004)
- F# (Syme, 2006)

LISP (1958)

First functional programming language

LISt **P**rocessing

Lots of **I**rritating **S**uperfluous **P**arentheses

- John McCarthy (MIT)
- *eager* evaluation
- impure features
  - assignment
  - dynamic binding (confusion between local and global variables)
  - `Quote` operator
  - fixed-point operator `LABEL` (implemented as cycle)

ML (1973)

First important typed FL

- Robin Milner (University of Edinburgh)
- *eager* evaluation
- implicit typing (Curry style)
- *types are automatically derived* (Hindley-Milner alg.)
- type-safe *exception handling*
- impure (assignments)

main additions to $\lambda \rightarrow$

- new primitives
  - fixed point combinator Y
  - arithmetic operators
- 'let' construction

  $\mathrm{let\ x\ be\ N\ in\ M\ end.}$

Robin Milner (1934-2010)

Haskell (1990)

Being Lazy with Class

- designed by committee (P. Hudak, J. Hughes, S. Peyton Jones, P. Wadler, . . .)
- *lazy* evaluation (non-strict)
- *parametric* type polymorphism (System F)
- *purely functional*
- *type classes*
- side-effects through *monads*

Where can you find FP languages?

- telecommunications: *Erlang* – Ericsson
- banking: Credit Suisse (F#), ABN (Haskell), Bank of America (Haskell)
- insurance: Grange Insurance (F#)
- web applications: Facebook (OCaml))
- verification: SLAM – Microsoft, ASTRÉE – Inria (both OCaml)
- user applications: Unison (OCaml)
- mathematical libraries: FFTW (OCaml)
- automated theorem proving: Coq (OCaml)
- development tools: Darcs (Haskell)

Functional features in imperative languages

- anonymous functions
  (JavaScript, Python, Ruby, C#)
- (some) higher-order functions
  (e.g. Python: `filter, map`)
  ```
  map(lambda x:  x ** 3, [2, 4, 6, 8])
  ```
- partial function application (Python)
  ```
  add5 = partial (add, 5)
  add5(15)
  ```
- lists (Python, C#, . . .),
  list comprehensions (Python)
- (type derivation (C# 3.0, C++11, Visual Basic 9.0))

Reading list

J. Hughes: *Why Functional Programming Matters*

H. Barendregt: *The impact of the lambda calculus in logic and computer science*

Cardone, Hindley: *History of Lambda-calculus and Combinatory Logic*