Termination

Typing

Grades

- 10% - participation & exercises
- 90% - term paper

- Alonzo Church (1903-1995)
  - invented lambda calculus (1932)
  - first programming-language researcher (sans computers)
  - Turing's advisor

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

By Alonzo Church.[2]

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.

free and bound variables

In consequence of this abstract character of the system which we are about to formulate, it is not admissible, in proving theorems of the system, to make use of the meaning of any of the symbols, although in the application which is intended the symbols do acquire meanings. The initial set of postulates must of themselves define the system as a formal structure, and in developing this formal structure reference to the proposed application must be held irrelevant. There may, indeed, be other applications of the system than its use as a logic.

symbols do not have pre-conceived meanings

In consequence of this abstract character of the system which we are about to formulate, it is not admissible, in proving theorems of the system, to make use of the meaning of any of the symbols, although in the application which is intended the symbols do acquire meanings. The initial set of postulates must of themselves define the system as a formal structure, and in developing this formal structure reference to the proposed application must be held irrelevant. There may, indeed, be other applications of the system than its use as a logic.

symbols do not have pre-conceived meanings

Proof terms, well-formed objects

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

Lambda Calculus

- Everything is a function
- For example, λx.x is the identity function
- λy.λx.x is a constant function, always returning identity

Lambda Terms

- Constants C; Variables X
- L = constant | variable | application | abstraction
- L ::= C | X | (LL) | λX.L

Positions

- Dewey decimal system
- Number children, left to right
- Path to position gives "address"

Free Occurrences

- Constants C; Variables X
- L := C | X | (LL) | λX.L
- $F_x(c) = \{\}$ $\quad\quad$ $F_x(x) = \{e\}$
- $F_x(st) = 0.F_x(s) \cup 1.F_x(t)$
- $F_x(\lambda x.s) = \{\}$
- $F_x(\lambda y.s) = 1.F_x(s)$

Lambda Calculus

- β-rule: $(\lambda x.s)\,t \rightarrow s[x \mapsto t]$

- Replace (all free) x in s with t

Substitution

- $x[x \mapsto t] = t$
- $y[x \mapsto t] = y$
- $c[x \mapsto t] = c$
- $(su)\,[x \mapsto t] = s[x \mapsto t]\ u[x \mapsto t]$
- $(\lambda x.s)\,[x \mapsto t] = \lambda x.s$
- $(\lambda y.s)\,[x \mapsto t] = \lambda y.\ s[x \mapsto t]$

Beta Immortality

- $\lambda x.x(x)\ \lambda x.x(x) \rightarrow \lambda x.x(x)\ \lambda x.x(x)$

Completeness

- Every recursive function can be simulated by a pure lambda expression.
- Church numerals represent the naturals.
- Termination is undecidable.

Church Numerals

- n
- $\lambda f,x.f^n(x)$

Church Numerals

- T
- F
- if(c,a,b)

- λx,y.x
- λx,y.y
- λc,a,b.c(a,b)

- 0
- n++
- n--
- n=0

- λf,x.x
- λf,x.f(n(f,x))
- hard
- n(λx.F,T)

Synagogue Numerals

- T
- F
- if(c,a,b)

- λx,y.x
- λx,y.y
- λc,a,b.c(a,b)

- O
- n++
- n--
- n=O

- λx.x
- λx.x(F,n)
- n(F)
- n(T)

Scheme

- (((lambda (x y) (y x)) (lambda (z) z) (lambda (z) (z z))) 5)
- (((lambda (z) (z z)) (lambda (z) z)) 5)
- (((lambda (z) z) (lambda (z) z)) 5)
- ((lambda (z) z) 5)
- 5

Inner vs. Outer

- Scheme uses innermost
- Haskell uses outermost

Recursor

- Y := (λx.(λy.x(y(y))) (λy.x(y(y))))
- Y(b) : recursive function with body b
- fixpoint: Y(b) = b(Y(b))
- (Y(λf.λm,n.if(n=0,m,(f(m,n--))++)))(3,4)

Currying

λx.λy.A[x,y] instead of λx,y.A[x,y]

+ is the binary addition function

+(3) adds 3 to any number

+(3)(4) evaluates to 7

Arithmetic (Rosser)

• 0	• $\lambda f.\lambda x.x$
• n++	• $\lambda f.\lambda x.n(f)\,(f(x))$
• m+n	• $\lambda f.\lambda x.m(f)\,((n(f))\,(x))$
• mn	• $\lambda f.m(n(f))$
• m^n	• $\lambda f.n(m)\,(f)$

λ-calculus and first-order rewriting led to two important families of programming languages:

- functional programming languages: Lisp (1958), ML (1972), Haskell (1990), OCaml (1996), F# (2005), ...
- rewriting-based languages: OBJ (1976), Elan (1994), Maude (1996), ...

Simple Types

- Base types B (e.g. Nat)
- Arrow types [e.g. Nat → (Nat → Bool)]
- Each constant/variable has a type
- $\mathrm{Type}(\lambda x{:}\sigma.\ s{:}\tau) \approx \sigma \to \tau$
- $\mathrm{Type}(s{:}\sigma \to \tau\ t{:}\sigma) \approx \tau$

Typing Rules

$$\frac{}{x : A \vdash x : A}\ \mathrm{Id}$$

$$\frac{\Gamma,\, x : A \vdash u : B}{\Gamma \vdash \lambda x.\, u : A \to B}\ \to\text{-I} \qquad \frac{\Gamma \vdash s : A \to B \qquad \Delta \vdash t : A}{\Gamma,\, \Delta \vdash s\, t : B}\ \to\text{-E}$$

Typed Lambda Calculus

- β-rule: $(\lambda x{:}\sigma.\ s{:}\tau)t{:}\sigma \rightarrow s[x{:}\sigma \mapsto t{:}\sigma]{:}\tau$

Typed Beta Mortality

- $\lambda x{:}\sigma\to\tau.\,(x{:}\sigma\to\tau\; x{:}\sigma) : (\sigma\to\tau)\to\tau$

Termination

- Turing gave first proof
- Tait's proof
  - Induction on term structure
  - Induction on type structure

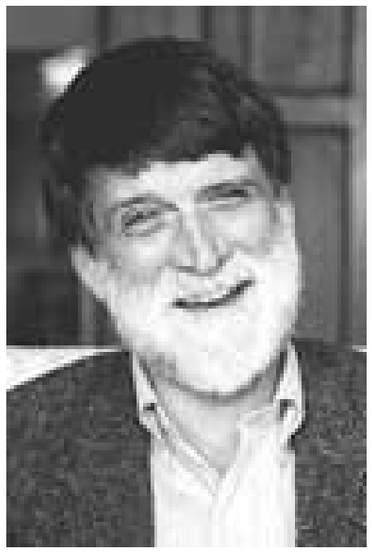
Termination of β-reduction alone?

in the simply-typed λ-calculus:

- $\to_\beta$ can be proved terminating by a direct induction on the type of the substituted variable (Sanchis 1967, van Daalen 1980)

  does not extend to rewriting where the type of substituted variables can increase, *e.g.* $\mathsf{f}(\mathsf{c}x) \to x$ with $x : \mathsf{A} \Rightarrow \mathsf{B}$

computability has been introduced for proving termination of β-reduction in typed λ-calculi (Tait, 1967) (Girard, 1970)



- every type T is mapped to a set $[\![T]\!]$ of computable terms
- every term $t : T$ is proved to be computable, *i.e.* $t \in [\![T]\!]$

Predicates

- S[t]: t is "terminating" (no infinite paths)
- C[t]: t is "computable" (typed terminating)
- N[t]: t is "normalizing" (has a normal form)

Facts

- $S[t] \;\&\; t \twoheadrightarrow u \Rightarrow S[u]$
- $S[t] \;\&\; t \triangleright u \Rightarrow S[u]$
- $\{\, \forall u.\, t \twoheadrightarrow u \Rightarrow S[u] \,\} \Rightarrow S[t]$

Desiderata

1. $C[t] \Rightarrow S[t]$

2. $C[s] \;\&\; s \rightarrow t \Rightarrow C[t]$

3. $C[x] \quad C[c]$

4. $\forall t \{ u(v) \rightarrow t \Rightarrow C[t] \} \Rightarrow C[u(v)]$

5. $C[u] \Leftrightarrow \forall v \{ C[v] \Rightarrow C[u(v)] \}$

Computability predicates

there are different definitions of computability (Tait Sat, Girard Red, Parigot SatInd, Girard $\mathrm{Bi}\perp$) but Girard's definition Red is better suited for handling *arbitrary* rewriting

let Red be the set of P such that:

- termination: $P \subseteq \mathrm{SN}(\to_\beta)$
- stability by reduction: $\to_\beta(P) \subseteq P$
- if t is neutral and $\to_\beta(t) \subseteq P$ then $t \in P$

neutral $=$ not head-reducible after application ($\lambda x u$ is *not* neutral)