λ-Calculus: The Other Turing Machine

Blelloch and Harper

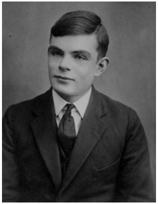
50th year celebration of CSD,
and 80th year celebration of Church and Turing

October 25, 2015

Church and Turing

In 1929-1932 Church developed the λ-calculus as a formal system for mathematical logic.

In 1935 he argued that any function on the natural numbers that can be effectively computed, can be computed with his calculus.



In 1935, independently, Turing developed what is now called the Turing Machine.

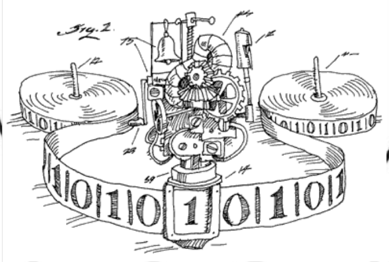
In 1936 he too argued that any function on the natural numbers can be computed with his machine.
He also showed the two models are equivalent.

The equivalence was a powerful indication of the "universality" of the models, and lead to what is now called the: "Church-Turing Thesis" (or "Church's law")

Church-Turing Thesis

$$(\lambda x.e_1)e_2 \Rightarrow_\beta e_1[e_2/x]$$

$$=$$



The "Church-Turing Thesis" is by itself is **one of the most important ideas** on computer science,
but the **impact** of Church and Turing's models **goes far beyond** the thesis itself.

Oddly, however, the impact of each has been in almost completely separate communities.

Turing Machine ⇔ Algorithms and Complexity
λ-Calculus ⇔ Programming Languages

The "Church-Turing Thesis" is by itself is **one of the most important ideas** on computer science,
but the **impact** of Church and Turing's models **goes far beyond** the thesis itself.

Oddly, however, the impact of each has been in almost completely separate communities.

Turing Machine $\Leftrightarrow$ Algorithms and Complexity
λ-Calculus $\Leftrightarrow$ Programming Languages

The impact and separation is not accidental.
"Two sources of beauty in programs: Efficiency and Structure"

Turing Machine ⇔ Algorithms and Complexity

Well suited for measuring resources (**efficiency**).

Ideas or fields developed from the Turing machine:

- Axiomatic complexity theory
- P vs. NP, polynomial hierarchy, P-space, ...
- RAM model and asymptotic analysis of algorithms
- Cryptography (based on hardness of computation)
- Learning theory (learning power of Turing machines)
- Algorithmic game theory
- Hardness of approximation

λ-Calculus $\Leftrightarrow$ Programming Language Theory

Well suited for composition and abstraction (**structure**).

Ideas or fields developed from the λ-calculus:

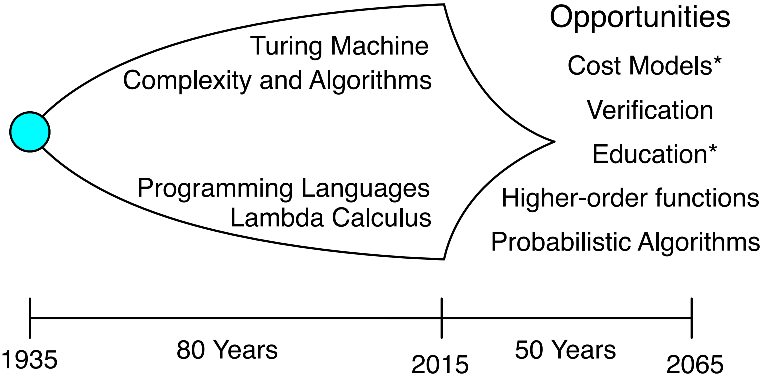
- Call-by-value, lexical scoping, recursion
- lambda, higher-order-functions (just now in C++ and Java)
- denotational semantics
- type theory (the theory of abstraction)
- implicit-memory management
- polymorphism
- proof-checkers: LCF, NuPRL, Coq, Isabelle
- Languages: Lisp, FP, ML, Haskell, Scala (Java, Python, C++)

The single most important change in Java 8 [Lambda Expressions] enables faster, clearer coding and opens the door to functional programming [Dr. Dobbs 2014]

The single most important change in Java 8 [Lambda Expressions] enables faster, clearer coding and opens the door to functional programming [Dr. Dobbs 2014]

Books on Amazon from past 10 years, with λ-calculus in title:

Opportunities



Problem with Cost Models

Has worked in the past for **algorithm design** because you can program an algorithm and then "compile" to the RAM in your head.

Problem with Cost Models

Has worked in the past for **algorithm design** because you can program an algorithm and then "compile" to the RAM in your head.

But: With new features programming languages are diverging from the machine-based cost models.

- parallelism, laziness, higher-order-functions, exceptions, memory management, built in aggregate types, ...

Problem with Cost Models

Has worked in the past for **algorithm design** because you can program an algorithm and then "compile" to the RAM in your head.

But: With new features programming languages are diverging from the machine-based cost models.

- parallelism, laziness, higher-order-functions, exceptions, memory management, built in aggregate types, ...

Claim: Analyzing costs directly on the RAM or any machine-models will fail in the long run. Wrong level of abstraction.
What is the option?

The λ-Calculus

Syntax: $e = x \mid \lambda x.e \mid e(e)$

Computation: repeat single rule called β-reduction:

$$\lambda x.[...x...x...](e_2) \Rightarrow [...e_2...e_2...]$$

Finished: when no expressions of the form $e(e)$

The λ-Calculus

Syntax: $e = x \mid \lambda x.e \mid e(e)$

Computation: repeat single rule called β-reduction:

$$\lambda x.[...x...x...](e_2) \Rightarrow [...e_2...e_2...]$$

Finished: when no expressions of the form $e(e)$

Example: $\lambda x.x(x)\ (\lambda x.x(x)) \Rightarrow \lambda x.x(x)\ (\lambda x.x(x))$

The λ-Calculus

Syntax: $e = x \mid \lambda x.e \mid e(e)$

Computation: repeat single rule called β-reduction:

$$\lambda x.[...x...x...](e_2) \Rightarrow [...e_2...e_2...]$$

Finished: when no expressions of the form $e(e)$

Example: $\lambda x.x(x)\ (\lambda x.x(x)) \Rightarrow \lambda x.x(x)\ (\lambda x.x(x))$

What about: recursion, conditionals, booleans, lists, trees, ...

Simple and efficient encondings.

Example of "Sugared" λ-Calculus

$$
\begin{array}{l}
\texttt{mergeSort}(A) \; = \\
\textbf{if} \; (|A| \leq 1) \; \textbf{then} \; A \\
\textbf{else} \; \textbf{let} \; (L, R) \; = \; \texttt{split}(A) \\
\quad \textbf{in} \; \texttt{merge}(\texttt{mergeSort}(L), \; \texttt{mergeSort}(R)) \; \textbf{end}
\end{array}
$$

Example of "Sugared" λ-Calculus

$\texttt{mergeSort}(A) =$
if $(|A| \leq 1)$ **then** A
else let $(L, R) = \texttt{split}(A)$
 in $\texttt{merge}(\texttt{mergeSort}(L),\ \texttt{mergeSort}(R))$ **end**

But what is the cost? Sequential and Parallel.

The λ-calculus does not define in which **order** to reduce.

Problem: does not make a good cost model because number of steps depends on the reduction order. And some orders are not efficient to evaluate (a single reduction could be expensive).

Virtue: it is inherently parallel. Church invented a parallel model!!!

The λ-calculus does not define in which **order** to reduce.

Problem: does not make a good cost model because number of steps depends on the reduction order. And some orders are not efficient to evaluate (a single reduction could be expensive).

Virtue: it is inherently parallel. Church invented a parallel model!!!

Key Idea:

1. Fix an order that is parallel, and cheap to evaluate.
2. Base a cost model on it.
3. Bound the cost when mapped to standard models.

Accounting for costs

Once we have an order, then we can:

1. count number of reductions (work)
2. count number of parallel steps (depth or span)

Bounded implementation

If w work and d depth in λ-calculus, then $O(w \log w)$ time on RAM, and $O((w \log w)/p + d)$ time¡ on PRAM with p processors.

Accounting for costs

Once we have an order, then we can:

1. count number of reductions (work)
2. count number of parallel steps (depth or span)

Bounded implementation

If w work and d depth in λ-calculus, then $O(w \log w)$ time on RAM, and $O((w \log w)/p + d)$ time¡ on PRAM with p processors.

Example:

$\texttt{mergeSort}(A) =$
if $(|A| \leq 1)$ **then** A
else let $(L, R) = \texttt{split}(A)$
  in $\texttt{merge}(\texttt{mergeSort}(L),\ \texttt{mergeSort}(R))$ **end**

Does $O(n \log n)$ work and has $O(\log^2 n)$ span.

Education

Based on this approach we (and others) developed two new introductory undergraduate classes:

15-150: Functional Programming
15-210: Parallel and Sequential Data Structures and Algorithms

- Over 300 students/year each.
- Teach parallelism from the start.
- Costs are calculated in terms of work and span.
- Algorithms are purely functional.

Conclusions

Next 50 years need to integrate Complexity/Algorithms and Programming Language Theory.

- Cost models based on languages, not machines. Particularly needed for parallelism.
- Other opportunities: Verification, type-theory and complexity, probabilistic programming, programs-as-data, cryptography and PL, game-theory and PL.