λ-Calculus: The Other Turing Machine

Guy Blelloch and Robert Harper

July 24, 2015

The early 1930s were bad years for the worldwide economy, but great years for what would eventually be called Computer Science. In 1932, Alonzo Church at Princeton described his λ-calculus as a formal system for mathematical logic,and in 1935 argued that any function on the natural numbers that can be effectively computed, can be computed with his calculus [4]. Independently in 1935, as a master's student at Cambridge, Alan Turing was developing his machine model of computation. In 1936 he too argued that his model could compute all computable functions on the natural numbers, and showed that his machine and the λ-calculus are equivalent [6]. The fact that two such different models of computation calculate the same functions was solid evidence that they both represented an inherent class of computable functions. From this arose the so-called Church-Turing thesis, which states, roughly, that any function on the natural numbers can be effectively computed if and only if it can be computed with the λ-calculus, or equivalently, the Turing machine. Although the Church-Turing thesis by itself is one of the most important ideas in Computer Science, the influence of Church and Turing's models go far beyond the thesis itself.

The Turing machine has become the core of all complexity theory [5]. In the early 1960's Juris Hartmanis and Richard Stearns initiated the study of the complexity of computation. In 1967 Manuel Blum developed his axiomatic theory of complexity, which although independent of any particular machine, was considered in the context of a spectrum of Turing machines (*e.g.*, one tape, two tape, or two stack). This was followed in 1971 by Stephen Cooke and Leonid Levin who showed a complete problem for NP (satisfiability) and introducing the question of whether P = NP. Since then a huge number of complexity classes have been isolated and related, including PCP, the class of Probabilistically Checkable Proofs. All this work has been defined in terms of minor variants of the Turing machine. In the field of Algorithms, where analysis needs to be more precise, other models have been developed, such at the Random Access Machine (RAM), that are closer to physical computers while remaining relatively abstract and hence widely applicable. However even these machines were heavily influenced by the Turing machine, and most of the complexity classes defined for the Turing machine carry over to the RAM.

Whereas the machine models became the foundation of complexity and algorithmic theory, it was Church's λ-calculus that became the foundation for the theory of programming languages [3]. This came about largely under the influence of Dana Scott and Christopher Strachey who, at the suggestion of Roger Penrose, developed denotational semantics, a topologically influenced account of higher-order computations acting on infinite data objects such as functions and streams, that are inherent in Church's formalism. Scott and Strachey's work meshed with Church's work on classical type theory as a foundation for mathematics, with L.E.J. Brouwer's program of constructive foundations that led to Per Martin-Löf's development of Intuitionistic Type Theory, and with N.G. de Bruijn's AUTOMATH language for expressing machine-checked proof, all of which were similarly founded on the λ-calculus. The result is an integrated theory of computation and deduction, known as the Propositions as Types principle, that consolidates programs with proofs, and types with propositions.

This principle lies at the heart of most programming language theory and many program verification and proof checking systems. Robin Milner's work on the LCF prover in the 1970's led to the emergence of functional programming, based directly on the λ-calculus, and interactive proof development, based on Scott's logic of computable functions arising from his program of denota-

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

Figure 1: Mergesort in a "sugared" λ-calculus. The algorithm is parallel since the recursive mergeSorts can run in parallel, and the merge itself can be parallelized. With built-in integers, and a parallel merge, it has $O(n \log n)$ work and $O(\log^2 n)$ span ($n = |A|$) [1].

tional semantics. The theory of polymorphism and data abstraction was developed by Jean-Yves Girard and John C. Reynolds based on Church's logical formalism. It is the single most important result supporting the modular design and development of large programs. Milner's and Reynolds's work not only influenced the design of the mathematically grounded functional languages such as ML or Haskell, but even the more prosaic languages such as C++ and Java, which have added λ-abstraction some eighty years after its invention. Building on Milner's LCF system most modern interactive provers, including NuPRL, Coq, Isabelle, and HOL, are based on the λ-calculus, the first two on constructive type theory, the latter two on classical type theory.

Despite the pervasive influence of Church's λ-calculus on programming, program verification, and mechanized mathematics, there has been surprisingly little overlap with the machine-based work on algorithms and complexity. Indeed, beyond Turing's initial paper, it is hard to find a research paper that even mentions both models. As a consequence, the subfields of the theory of programming languages and the theory of algorithms and complexity seem to be further apart than just about any other two subfields of computer science.

One natural question that arises is whether the λ-calculus can even be used for complexity theory and algorithm analysis. Here we point out that it can, and indeed using the λ-calculus offers some potential advantages. Whereas all other models of computation are based on the "program acting on data" paradigm, the λ-calculus is distinctive in that it consolidates the two; there is no distinction between program and data at all. What, then, does it mean to run a program, and, more to the point, what is the cost of doing so?

To answer this, we first briefly review the λ-calculus. Its syntax consists of the simple recursive grammar:

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

where x ranges over a set of variable names. It has one computational rule, called β-reduction, which takes an expression of the form $(\lambda x.e_1)(e_2)$ and substitutes e_2 for all the occurrences of the variable x in e_1, much as in high school algebra. A computation finishes when there are no reductions left to do. A time-based cost for the λ-calculus can be defined as the number of reduction steps it takes to convert an expression into its final form. This number, however, can depend on the order in which reductions are made. The λ-calculus admits several useful reduction strategies, some of which provide a deterministic model of computation with a well-defined cost measure based on reduction steps. Here we consider call-by-value (CBV) reduction, which starts by evaluating the top-level expression. If it is of the form $\lambda x.e$ then the evaluation is complete, otherwise the expression is of the form $e_1(e_2)$ and e_1 and e_2 are recursively evaluated, β-reduction is applied to the results, and the resulting expression is then evaluated. CBV corresponds to evaluating the argument, substituting it into the function body, and evaluating the body.

How does this cost measure based on the CBV reduction order for the λ-calculus compare to costs on the RAM model? It turns out that the two models are equivalent within a logarithmic factor [1]. In particular an expression evaluating with t reductions in the CBV λ-calculus can be simulated to run in $O(t \log t)$ time on the RAM, and a computation of input size n and taking t time on the RAM can be simulated with $O(t \log(t + n))$ reductions in the CBV λ-calculus.[1] This might seem surprising given that the λ-calculus does not have any control structures, does not support random access, and does not have any data types, not even Booleans or integers. The simulation works, however, since Booleans and various control structures can easily be supported with constant overhead, and random access and integers can be simulated with logarithmic overhead. If the λ-calculus is extended with built-in integers, then many algorithms have the same cost in the λ-calculus as in the RAM—for example merge sort (Figure 1) takes $O(n \log n)$ reductions. What the results say is that steps on the RAM and reductions in the CBV λ-calculus are surprisingly close, indeed much closer than steps on a RAM to steps on a Turing machine (which differ by a polynomial factor). The relationship means, for example, that the λ-calculus can easily be used to define complexity classes such as P, NP, EXP, or NEXP.

More importantly than the relationship of the CBV λ-calculus to the RAM is the fact that it is inherently parallel. This is because reductions can proceed in parallel. In particular, when evaluating an expression $e_1(e_2)$, the subexpressions e_1 and e_2 can be evaluated in parallel. In merge sort, for example, the recursive calls may be made in parallel. But, how do we assign costs to parallel evaluation? It turns out that using two cost measures—the total number of reductions (*work*) and depth of dependences of the reductions (*span*), are adequate. Both these costs can be computed compositionally—the work of evaluating e_1 and e_2 in parallel is the sum of the work of each, and the span is the maximum of the span of each. One can show bounds that relate these costs with time on parallel machine models. More recently the authors have investigated how the λ-calculus can also account for locality in functional programming [2]. Since there is no way in the λ-calculus to lay out memory, it is not immediately obvious that this is possible.

The λ-calculus therefore has some potential advantage over the Turing machine or RAM as a cost model—it does not separate functions from values, it is closer to programming languages (needing minimal "sugar" or extensions), and it is parallel. Given these advantages, here at Carnegie Mellon we use the λ-calculus (a sugared version) for analyzing costs in the new freshman-level functional programming class (15-150) and the new sophomore-level introduction to data structures and algorithms course (15-210). Both teach parallelism from the start.

We believe our work has just touched on potential interactions between the Church-ites and the Turing-ites, and believe there would be large benefit for Computer Science as a whole for the communities to interact more.

References

[1] Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In *ACM Conf. on Functional Prog. Languages and Computer Architecture (FPCA)*, pages 226–237, 1995.

[2] Guy E. Blelloch and Robert Harper. Cache efficient functional algorithms. *Commun. ACM*, 58(7):101–108, June 2015.

[3] F. Cardone and J. R. Hindley. λ-calculus and combinators in the 20th century. In D. M. Gabbay and J. Woods, editors, *Logic from Russell to Church*, Handbook of the History of Logic. North-Holland, 2009.

[4] Alonzo Church. An unsolveable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936. (Presented April 19, 1935).

[5] L. Fortnow and S. Homer. A short history of computational complexity. *Bulletin of the European Association for Theoretical Computer Science*, 80, June 2003.

[6] Alan M. Turing. On computable numbers with an application to the *Entscheidungsproblem*. *Proc. of the London Mathematical Society*, s2-42(1):230–265, 1937. (Presented Nov. 12, 1936).

[1]This makes the standard assumption that the word size of a RAM is at most $O(\log n)$ bits.