Gödel’s system $\mathcal{T}$ revisited. (English summary)


One of the goals of theoretical computer science is to study the expressive power of models of computation. One such model is $\lambda$-calculus, which is as expressive as Turing machines.

Interesting restrictions of $\lambda$-calculus exist. One is *typable $\lambda$-calculus* (TLC). TLC contains all, and only, those $\lambda$-terms to which we can assign formulas (of the implicative fragment) of intuitionistic logic as types. TLC rules out paradoxical behaviors existing in (full) $\lambda$-calculus. So, every term we can write as a term of TLC terminates, a feature that computational models as powerful as Turing machines cannot have.

The expressive power of TLC is sufficiently high to represent a fairly large set of numerical functions [H. Schwichtenberg, Arch. Math. Logik Grundlagenforsch. **17** (1975/76), no. 3-4, 113–114; MR0416864 (54 #4928); H. Simmons, Bull. Symb. Logic **11** (2005), no. 3, 321–350; Zbl 1096.03049]. So, in principle, TLC can be used as a paradigmatic functional programming language. However, its extremely essential syntax requires one to encode every data structure—even integers—from scratch. So, it has been natural to formulate paradigmatic languages with primitives that can be encoded as terms of TLC, but which make “programming” less tricky. Gödel’s system $\mathcal{T}$, which extends TLC with numbers and a recursion operator, is one of them.

On the other hand, the advent of linear logic [J.-Y. Girard, *Theoret. Comput. Sci.* **50** (1987), no. 1, 101 pp.; MR0899269 (89m:03057)] taught us how to refine the logical connectives of intuitionistic logic. This had a basic impact on the way we could look at TLC terms. Specifically, we can always see them as *linear $\lambda$-calculus* terms interacting with operators that manipulate arguments non-linearly. By definition, every linear $\lambda$-calculus term is syntactically linear. Namely, 1 is the number of occurrences of every variable in a given linear term.

Linear $\lambda$-calculus is relatively weak. The cost of deciding whether two of its terms evaluate to the same result is PTIME-complete, as linear $\lambda$-calculus can encode all Boolean functions [H. G. Mairson, J. Funct. Programming **14** (2004), no. 6, 623–633; Zbl 1063.68036].

The paper under review supplies a possible answer to the following question: We know we can adopt Gödel’s system $\mathcal{T}$, which can be viewed as a least extension of TLC, as a paradigmatic functional programming language. At the core of TLC is linear $\lambda$-calculus. Is there any least extension of linear $\lambda$-calculus we can see as the core of functional programming languages too?

The answer is positive thanks to System L, which the paper defines. System L is linear $\lambda$-calculus with natural numbers, and an iterator able to iterate closed linear terms only. We recall that a closed term is, in this context, a program whose meaning is completely self-explanatory and does not depend on the context.

System L is as expressive as Gödel’s system $\mathcal{T}$ when evaluating its terms under a closed reduction strategy. “Closed” means that the main computational steps—$\beta$-reduction and the application of
the System L iterator—can be triggered only if their arguments are closed (in the above sense).

The authors leave a natural extension of the question that they answer as an open problem: Is there any System PCFL, based on closed reductions, that captures the computationally complete functional programming language PCF, exactly as System L captures Gödel’s system T?

To my understanding, another potentially interesting research direction is connected to the area of implicit computational complexity (ICC), which aims at characterizing computational complexity classes by formal tools other than Turing machines which are endowed with a clock that bounds the time within which a result must be output. Questions worth answering might be: Is there any restriction of System L able to express functional programs with polynomial complexity only? If any such system could exist, what has it to do with proposals like those in [S. J. Bellantoni, K.-H. Niggl and H. Schwichtenberg, Ann. Pure Appl. Logic 104 (2000), no. 1-3, 17–30; MR1778933 (2001j:03079)] and [U. Dal Lago, S. Martini and L. Roversi, in Types for proofs and programs, 178–193, Lecture Notes in Comput. Sci., 3085, Springer, Berlin, 2004; MR2151334], among others?

Answering such questions could contribute both to ICC and to understanding where the expressiveness of System L comes from, despite the fact that it looks like an innocuous extension of linear λ-calculus.

Reviewed by Luca Roversi

References

27. J. Lambek, From lambda calculus to cartesian closed categories, in: J.P. Seldin. J.R. Hindley


Note: This list reflects references listed in the original paper as accurately as possible with no attempt to correct errors.

© Copyright American Mathematical Society 2011, 2013