Introducing: Reasonable Complete Programming Languages

Amir M. Ben-Amram*

December 7, 1996

Programming languages are the tool that computer programmers use for expressing algorithms. In contrast, Theoretical Computer Science uses many different kinds of formalisms for expressing algorithms: in particular, abstract machines of different structures, logical circuits and some formal calculi. In fact programming languages have played a minor role in textbooks and articles on the theory of computation. But this tendency has changed in recent years, and a growing number of authors put an emphasis on the programming language as a computing formalism that can be conveniently used in theoretizing about computation. This choice is important for various reasons, among which a major one is tying together the two major branches of Computer Science: the theory of computation (computability and complexity) and the theory of programming languages and program construction.

Needless to say, there are many different programming languages, in fact they probably outnumber the other, more abstract, models of computation. But according to the *Church-Turing Thesis* there is a common upper bound to their computational capabilities: any function that can be programmed in a viable programming language would also be Turing computable. Furthermore, it is "common wisdom" that any general-purpose programming language is able to simulate a Turing machine, that is, it achieves the "upper bound". A formal term for this property exists in theoretical literature: we call it *Turing completeness*. So, more formally stated, the "common wisdom" is that typical programming languages are Turing complete.

The importance of this statement is that it allows us to abstract from the particular language, and state results on computability in a general fashion. This relates to results which say that a particular function is computable as well as those that say that it is not.

I would like to argue that "typical" programming languages share additional properties, significant from a computability-theoretic point of view, which are not captured by the property of Turing completeness. For example, we may expect a programming language

*Author's address: The Academic College of Tel-Aviv–Jaffa, 4 Antokolski Str, 64044 Tel-Aviv, Israel; e-mail: `amirben@server.mta.ac.il`

L to have the *universal program property*: namely, it should be possible to write an L-program that on input (p,d) runs L-program p on data d (in programming terms, this is a *self-interpreter*). Such a property does not follow from Turing completeness: the latter says too little about the language. Another example of a statement which we should expect to be of general validity is "the halting problem for programs in a Turing-complete language is undecidable". Once again, proving this requires knowledge of other properties of the language besides what functions it can compute (the reader might try to identify such assumptions by redoing Turing's proof).

I suggest that it is desirable to adopt a definition for a class of "reasonable" programming languages; one which will allow general statements as above to be formalized and proved. This class should provide the "right" level of abstraction for reasoning about computability in programming languages.

Here is, then, my proposition for defining such a class. Let I be a chosen fixed Turing-complete language (for concreteness, this could be the language given under this very name in [1]) [1]. A language L will be called *reasonable* if there is a compiler from L into I, written in I.

Let us call L a *reasonable complete language* if, in addition to the above, there is a compiler from I into L (also written in I).

The main goal of this note is to argue that the above definition is "right". First, to defend the term "reasonable", I claim that in reality, we would not consider a proposed language reasonable if we cannot compile it into, or interpret it in, some computer's assembly language (or any other existing language). Since the I language is equivalent to an idealized "assembly language" (the language of an idealized RAM), our definition of "reasonable" matches this consideration.

Now to the theoretical benefits. Here is a sample of statements which are all true if L is a reasonable complete language.

1. There is a universal program in L.

2. L satisfies Kleene's s-m-n theorem. Fully stated: there is an L program spec such that for every L program p, if p computes the function $f(x, y)$ (formally: $[\![\mathtt{p}]\!] = \lambda xy.f(x, y)$) then $[\![\mathtt{spec}]\!](\mathtt{p}, \mathtt{d})$ is a program that computes $\lambda y.f(\mathtt{d}, y)$.

3. The halting problem for L programs is undecidable.

[1] To the readers not familiar with Jones' book, I would like to emphasize that the language I can be chosen for the definition in many other ways. For example, it could be a language for describing Turing machines. The advantage of Jones' language is, that (like actual programming languages) it is defined together with a concrete syntax for representing programs, while Turing machines and other theoretical formalisms are abstract objects. The concrete representation is necessary for allowing compilation from or to the language.

Proofs: all these statements are proved in [1] for the language `I` (and in many other books for other, equivalent languages). Using the existence of compilers to and from `I` to establish the statements for `L` is an easy exercise. I invite the reader to spell one such proof out in detail: it will become clear why the existence of *compilers* was required and not mere equivalence of computing power.

Readers familiar with abstract recursion theory (as in Rogers [2]) will recognize that the first two statements above, combined with Turing completeness and the assumption that the language is not more powerful than a Turing machine, give the set of properties that define an "acceptable enumeration of partial recursive functions". By using this notion, Rogers was able to develop recursive function theory in the most general fashion. One can say that the concept of a "reasonable complete language" is a programming-language equivalent of Roger's notion. However, I believe that the definition given above is a natural answer to the question "what constitutes a reasonable language", and it is better to have a result such as the existence of a self-interpreter as a theorem rather than a prerequisite.

I would also like to mention that reasonable languages (without the completeness requirement) are also an important class. This includes languages which, intuitively, can be implemented, whether they are Turing-complete or weaker. An example statement that can be proved for such languages is: *if a reasonable language computes only total functions, it cannot compute all the Turing-computable ones.* Note that it is not hard to invent languages which express precisely the total computable functions, but then they will not be reasonable in our sense, i.e., they would not be akin to implementation.

A final technical remark: the definition of "reasonable complete language" is not the conjunction of those of "reasonable" and "complete". The reason is that we understand the expression "`L` is complete" in the traditional way (`L` can compute every `I`-computable function), without assuming the existence of a compiler.

References

[1] N. D. Jones, *Computability and Complexity — from a Programming Perspective*, The MIT Press, Boston, 1997.

[2] H. Rogers Jr., *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, 1967.