Constructive Mathematics and Automatic Program Writers

by

Robert L. Constable

Technical Report
No. 70-86
November 1970

Department of Computer Science
Cornell University, Ithaca, N. Y.

Constructive Mathematics and Automatic Program Writers

by

Robert L. Constable*

Abstract

One point made here is that formal constructive mathematics can be interpreted as a "high-level" programming language; another point is that there are good reasons for doing so. Among them is the fact that a theoretical basis for automatic program writers (APW's) becomes especially perspicuous (in such a context the problem of assigning meaning to programs a la Floyd [6] is the inverse of program writing). Another reason is that such an interpretation reveals a number of interesting mathematical problems in the theory of computing.

While making these points we find occasion to present new observations on the completeness and efficiency of automatic program writers and to formulate a specific example of what we call von Neumann's principle on the logical complexity of systems. We apply the principle in the automatic program writing context and discuss its more general ramifications about the intelligibility of programs.

§I Basic Computing Systems

Consider the following models of computing.

1.1 Register machines, either RAM's (Random Access Machines) as in Shepherdson & Sturges [11] or RASP's as in Elgot & Robinson [5]. Associated with each such machine class is its machine language. In the case of [11] the absolute addresses are $\aleph = \{0,1,2,...\}$ and the basic instructions are A(n), increase contents of register n by 1, D(n), decrease contents of n if positive and

---

* Department of Computer Science, Cornell University, Ithaca, New York.

[illegible] in part by the National Science Foundation under

C(n,p) go to instruction p if contents of n are non-zero.

1.2 Programming languages, like GR or G_3 of Constable & Borodin [3] or languages like reference Algol. (A language like G_3 can be regarded as an assembler language for a RAM.)

1.3 Functional calculi, like Kleene's μ- calculus, [8], based on primitive recursive and the least number operator, μ. A richer example of such a calculus would be Kalmar's calculus for $\mathcal{E}$, the elementary functions, augmented by primitive recursion (1-fold recursion) and the μ- operator. Call such a calculus, Δ_1.

The functional calculi are intermediate between programming languages and mathematical theories. They allow specification of algorithms, but they suppress the "programming details." More precisely there is a natural mapping, T, of Δ_1 into GR such that e and T(e) both express the same function, but the map is not unique because there are numerous ways to translate the mathematical expressions into algorithms. (The translator T behaves like a "compiler" but is higher level and so might be called a meta-compiler.)

1.4 Formal first order constructive theories of arithmetic, such as Kleene's Intuitionistic Number Theory in IM [8], denoted INT.

It is not commonly recognized that such theories can serve to specify algorithms for function, $f: \aleph^{\rho} \to \aleph$. The theorem which makes this possible

1.5 (Kleene) If $F(x_1,\ldots,x_n,y)$ is a formula in INT and $\forall x_1 \cdots \forall x_n \exists! y$ $F(x_1,\ldots,x_n,y)$ is provable, then there is a unique recursive function f() such that

$$F(x_1,\ldots,x_n, \;\; f(x_1,\ldots,x_n))$$

and a program, f, for f() can be determined effectively from a proof of F.

In particular then, there is a translator W: INT $\rightarrow$ Algol. More precisely, given any formula F expressing a function in INT and a proof, p_F, of

$$* \quad \forall x_1 \cdots \forall x_n \exists y \;\; F(x_1,\ldots,x_n,y)$$

$W(\langle F,p_F\rangle)$ is an Algol program computing the f of 1.5.

1.6 We call W a <u>program writer</u> (PW) to indicate the "high-level" nature of the translation. It is interesting to note that the "mathematical content" of the function f is expressed by F and that in a sense we have suppressed the computational detail at the mathematical level. An important consequence of this for mathematics is that one can concentrate on the logical structure of the function and seek conceptually simple existence proofs and equivalence results. Such results often are poorly related to good computation, e.g., efficient or short programs, but hopefully are exemplification of good logic, e.g., elegant or general proofs.

§2 Automatic Program Writers

One basis for automatic program writing is a translator like W. To automate the program writer W, one builds an <u>automatic theorem prover</u> A and sets for it the task of proving statements like * about formulas like F. Waldinger [14] has considered systems of this type based on classical number theories. The classical approach requires a restriction on the type of proof procedure because not all classical proofs can be translated into programs.

2.1 A first step in this approach to automatic program writing would be the implementation of the program writer W. We will omit the details of our version of W deferring them to a later report. Instead we concentrate on certain theoretical questions about W.

§3 Completeness of Program Writers

3.1 A reasonable definition of completeness for a program writer is that it be able to write a program for every recursive function, i.e., map formulas and proofs into $\mathcal{R} = \{$all recursive (total) functions $f: \mathbb{N}^p \to \mathbb{N}\}$. Waldinger [14] aims at a completeness proof for his classical system, but to get one he changes his PW from mapping a formula F and a proof of $\forall x\, \exists!\, y F(x,y)$ to a program, F, into a scheme for <u>numeralwise representability</u>, i.e., finding a proof for $\exists!\, yF(\underline{x},y)$ for a specific numeral $\underline{x}$. The proof, say p_x, leads to a computation for f(x), but it does not produce a program f which is valid for all imputs. We will call this a representation theorem for $\mathcal{R}$ in number theory, not a completeness theorem for a PW. In fact, we note that a PW as we define it cannot be complete because it produces an r.e. subset of .

3.2 Let T be a constructive first order number theory and let $\mathcal{A}$ be a universal algorithmic language. If W: T $\to \mathcal{A}$ is a program writer in the sense of 1.6, then W is incomplete wrt $\mathcal{R}$, i.e., $\exists$ $g \in \mathcal{A}$ g computes $g() \in \mathcal{R}$ and there is no formula G and proof p_G such that $W(\langle G, p_G \rangle) = g$.

This lack of completeness is essential because there are not enough proofs in T to prove all formulas F which define total functions. But the result might not be desturbing if W wrote all "practical programs". Indeed one might argue that since for a reasonable theory T, like INT, W can write programs for all primitive recursive functions, $\mathcal{R}^1$, and $\mathcal{R}^1$ certainly includes all practical functions, that is, W is "complete enough". But something else might happen. Perhaps W cannot produce all of the desirable programs for $\mathcal{R}^1$. We find that this is indeed the case.

3.3 Let $\mathcal{R}_T$ be the set of all (recursive) functions definable in T. We can take it to be those computed by the programs in W(T). Call these the <u>W-programs</u> of $\mathcal{A}$.

What we have argued above is that $\mathcal{R}^1 \subseteq \mathcal{R}_T \subset \mathcal{R}$ if T includes INT.

Applying Blum's theorem on the size of programs in subrecursive formalisms, (see [2] and [4]), we can conclude that

3.4 There are functions in $\mathcal{R}_T$ whose shortest W-programs are arbitrarily longer than one of their non-W $\mathcal{A}$ programs, i.e., for any $f() \epsilon \mathcal{R}$ let ω_i denote W-programs in $\mathcal{A}$, and let $|\alpha_i|$ denote the size of any $\mathcal{A}$ program according to Blum [2], then for every n there is an ω_i , $|\omega_i| \geq n$, and if $\omega_j() = \omega_i()$ then $|\omega_i| \leq |\omega_j|$ and there is an $\alpha_1() = \omega_i()$ and $f(|\alpha_i|) < |\omega_i|$.

To see the import of this, take $f(x) = 2^{2^{2^x}}$. Therefore some W-programs are arbitrarily uneconomical. The compensatory virtue of these programs is that we know that they correctly compute the mathematical function defined by the formula F. Therefore we need not consider <u>correctness proofs</u> for them.

One might conjecture that it will be difficult to verify the correctness of the programs like α_i in 3.4. In a sense this is correct because the size of the "verification" will be very large in comparison with the size of the program. Such a phenomenon can be regarded as an illustration of the following principle articulated by von Neumann in [13], p. 47.

3.5 von Neumann's Principle: "... when you get to high complications, the actual object is simpler than the literary description."

3.6 As a specific version of the principle we take a verification of α to be a formula F_i in INT which describes α and a proof that F_i is correct for α.

Statement 3.4 suggests that small programs can have arbitrarily large descriptions, but 3.4 does not prove this since a program writer like W is only one way to relate a formula F_α to α.

3.7 Suppose we look at the inverse of the program writing process. Namely, given $\alpha_i \in \mathcal{A}$, assign to α_i a formula F_i which describes α_i and whose proof verifies α_i. One approach to this is Floyd's [6]. To accomodate it we must make it clear how programs are to be described in a numerical language like INT. But this is simply any standard encoding of the semantics of into INT.

The verification mapping $V:\mathcal{A} \to T$ cannot verify all programs (for instance those programs in $\mathcal{A}$ with no equivalents in W(T)) and therefore cannot be automatic. However, from the standard proofs in recursive function theory we know that there is an algorithm which will assign a descriptive formula F_i to α_i which numerically represents α_i (see 3.1).

We can show that even the best verifier confirms our suspicions of 3.4. Namely,

3.8 Given any $f() \in \mathcal{R}$ and any n there is a function $\alpha_i()$ in $\mathcal{R}_T$ such that $|\alpha_i| \geq n$ and the shortest verification of α_i in T is greater than $f(|\alpha_i|)$.

The 3.8 version of von Neumann's principle leads to some interesting speculation about ultra complex programs, like operating systems or theorem proving systems. It says that is is possible to build a system which is so logically complex that there is no hope of ever understanding it although it is quite feasible to build (say for instance that $|\alpha_i| = 10{,}000$ and $f(x) = 2^{2^x}$). It seems that such complexity has already been achieved by existing highly complex systems, and one can argue that our only hope of understanding them is writing them from the high level down. We see from 3.4 that we might pay a high price in size for this intelligibility The practical question probably becomes a matter of balancing the parameters. Another important parameter to consider is efficiency. We turn to it next.

§4 Efficiency

4.1 It is natural to ask how efficient the W(-T) programs are for $\mathcal{R}_T$. Conceivably there exist programs in $\mathcal{A}$ which compute $\mathcal{R}_T$ functions arbitrarily better than W-programs. That is, for all $h \in \mathcal{R}_T$ there exists $f() \in \mathcal{R}_T$ such that if $\omega_i() = f()$ then there is an $\alpha_i() = f()$ such that $g(m\alpha_i(x),x) < m\omega_i(x)$ a.e.x where $\{m\alpha_i\} = \Phi$ is a Blum complexity measure, [1]. This would say that "W produced arbitrarily 'bad' code."

4.2 We can show that for reasonable examples of $T, \mathcal{A}$, and Φ such as INT and Algol or RASP or RAM machine language, where Φ is time, there is a fixed inefficiency factor h(). That is, within a modulus of h() around the computational complexity of $\mathcal{A}$ program for an $\mathcal{R}_T$ function there is a complexi function, $m\omega_i()$, for a W-program for that function. Unfortunately our version of the proof is quite complex, relying on subrecursive hierarchy theory The same principles apply in the case of $\mathcal{R}^1 \subset \mathcal{R}_T$ and the proof procedure is more transparent. (For a look at this problem for $\mathcal{R}$ see [7].)

4.3 Let T be INT and $\mathcal{A}$ be GR and $\{t\alpha_i\}$ the time measure on $\mathcal{A}$. There is an $h() \in \mathcal{R}^1$ such that if $\alpha_i() \in \mathcal{R}^1$, then there is a W-program ω_i such that

$t\omega_i(x) \le h(x, t\alpha_i(x))$ a.e.x.

The idea is simply that any $\mathcal{A}$ program, α_i, can be simulated by a program in normal form with a bounded μ-operator, say $\alpha_i(x) = V(\mu y \le f_n(x)\ T(i,x,y))$ where $\{f_n\}$ is a spine (see [4]) for $\mathcal{R}^1$. This canonical form say η_i, is a W-program since it can be defined by primitive recursion. The run-time, $t\alpha_i()$, is within a fixed simulation cost of $t\eta_i()$, so the theorem holds.

To extend such proofs to $\mathcal{R}_T$ we need a more detailed analysis of $\mathcal{R}_T$ which in turn depends on information about T. When T is INT the results hold, but the proofs are difficult. It would be nice to have a smooth running general theory for questions of this type (general in T, $\mathcal{A}$ and the measure $\{m\alpha_i\} = \Phi$). Examples of the existing (cumbersome) theory can be seen in Constable [4] where there are further references to the work of J. Robbin, L. Bass, P. Axt, and others.

Results about efficienty and size would be helped considerably by any of the following theoretical advances:

(1) Development of an elegant way to represent partial functions by formulas in T.

(2) Development of a natural mapping from programs in $\mathcal{A}$ to their descriptions in T.

(3) Characterization of the concept of "higher" and "lower" level computing models and of the "natural" translations from level to level.

(4) Characterization of the natural computational complexity measures on subrecursive classes like $\mathcal{R}_T$ or $\mathcal{R}^1$ (see [4]).

Some specific interesting questions are:

(1) Is there an automatic verifier for W-programs such that

$W(V(\alpha)) = \alpha$

(2) What is the best modulus of efficiency, h(), for a reasonable W mapping INT into Algol?

The author would like to thank John Cherniavsky and Professor Robert Wagner of Cornell University for their helpful discussions of the ideas in this paper.

References

[1] Blum, M., Machine-Independent Theory of the Complexity of Recursive Functions, Journal of the Association for Computing Machinery, 14, 1967, p. 322-36.

[2] Blum, M., On the Size of Machines, Information & Control, 11, 1967, p. 257-265.

[3] Constable, Robert L. and Borodin, Allan B., On the Efficiency of Programs in Subrecursive Formalisms, IEEE Conference Record, Symposium on Switching & Automata Theory, 1970. (To appear in JACM).

[4] Constable, Robert L., On the Size of Programs in Subrecursive Formalisms, ACM Symposium on the Theory of Computing, 1970, p. 1-9.

[5] Elgot, C., and Robinson, A., Random-access Stored Program Machines, an Approach to Programming Languages, Journal of the Association for Computing Machinery, 11, 1964, p. 365-399.

[6] Floyd, Robert W., Assigning Meaning to Programs, Proceedings of Symposia in Applied Mathematics, Vol. XIX, American Mathematical Society, 1967, p. 19-32.

[7] Hartmanis, Juris and Constable, Robert, Complexity of Formal Translation and Speed-up Results, Computer Science Department, Cornell University, Technical Report 70-85 , 1970.

[8] Kleene, S. C., Introduction to Metamathematics, Princeton, 1952.

[9] Manna, Z. and Waldinger, R., Towards Automatic Program Synthesis, (preliminary draft, personal communication), 1970.

[10] Rogers, H., Theory of Recursive Functions and Effective Computability, New York, 1967.

[11] Shepherdson, J. C. and Sturgis, H. E., Computability of Recursive Functions, Journal of the Association for Computing Machinery, 10, 1963, p. 217-255.

[12] Schwartz, J. T., Set Theory as a Language for Program Specification and Programming, (preliminary draft), New York University report, 1970.

[13] Von Neumann, J., Theory of Self-Reproducing Automata, University of Illinois, Urbana, 1966.

[14] Waldinger, Richard, Constructing Programs Automatically using Theorem Proving, Technical Report, Department of Computer Science Carnegie-Mellon University, 1969.