Programs with infinite loops: from primitive recursive predicates to the arithmetic hierarchy

((quite) preliminary)

Armando B. Matos

September 11, 2014

Abstract

Infinite time Turing machines have been studied by Hamkins, Loöwe, Welch and others. In this work we concentrate in a restricted form of infinite programs ("iprogrs"). These programs read integers (the input arguments) and output an integer. They always halt, when "halting" is defined in terms of the "limit technique". After the execution of an infinite loop, the value of a programming variable x is defined to be $\lim_{i\to\infty} x_i$ (if the limit exists), where x_0, $x_1\ldots$ is the sequence of values taken by x. Most previous work on infinite time Turing machines uses instead $\limsup x_i$ (which is always defined).

The language of iprogrs is as an extension of the LOOP language, which characterizes exactly the class of primitive recursive functions. The language of iprogrs characterizes the arithmetic hierarchy. Simple syntactic restrictions based on the maximum depth of infinite loop nesting correspond exactly to the levels of the arithmetic hierarchy. Some considerations on the analysis of the complexity of these infinite programs are also included in this work.

1 Introduction

We introduce the concept of *iprogrs*, or "infinite programs". Imagine that each basic step of a program runs 10% faster than the previous one. Then, infinite loops finish in finite time, that is, an infinite number of program steps is executed in a finite time. In this note we study a sub-class of these infinite programs, or *iprogrs*. This sub-class, which can be seen an extension of the LOOP language[1], is sufficient to solve (in infinite time) every problem in the arithmetical hierarchy.

We show that all first order logical statements are "solved" by these infinite programs, see an example in Figure 1, and that specific levels of the hierarchy correspond exactly to syntactic restrictions of iprogrs.

As we are considering the execution of infinite loops, we will use sentences with unorthodox and strange meaning, like: "when the (infinite) loop finishes" "after the execution of the infinite loop", the program (containing infinite loops) "solves" the problem... In order to avoid confusion we will quote these unorthodox uses (see the Section "About infinite time programs" below) , writing for instance, every iprogr "halts".

[1]The LOOP language characterizes exactly the set of primitive recursive functions, see below.

```
Input: the index m of a TM and its input x
    c ← 0
    Initialize the computation of φ_m(x);
    for all x do
          if c = 0 then run one step of the computation φ_m(x);
          if the TM halted then c ← 1
    done
    return c
```

Figure 1: An infinite program that solves the halting problem.

About infinite time programs

In order to represent infinite programs we use a computation model based on a register programming language because (i) as said above, primitive recursive functions are easily characterized by a very simple register programming language LOOP [7, 8] (whose programs always halt) and (ii) due to the Kleene normal form Theorem, the arithmetic hierarchy can be bootstrapped (the level 0 of the hierarchy) with the primitive recursive predicates. Our language for infinite programs is obtained by adding to the instructions of LOOP a single form of infinite loop.

Consider the sequence of values taken by the programming variable x, say x_0, $x_1 \ldots$. If the limit $x' \stackrel{\text{def}}{=} \lim_{i\to\infty} x_i$ exists, and, after the loop, we only use the values of variables whose sequence has a limit, then there is some time t_0 after which x has the constant value x'.

Consider an infinite loop and programming variable whose sequence of values (associated with that loop) has a definite limit. The infinite loop can be "breaked" after a long enough time so as to guarantee that the sequence has already reached the final (integer!) value.

It is illustrative to reach the same result – the possibility of breaking infinite loops – using the concept of Kolmogorov complexity [5]. Consider the execution of a program containing the infinite loop

$$c \leftarrow 0;\ \texttt{for all}\ x\{\texttt{if}\ f(x) = 0\ \texttt{then}\ c \leftarrow 1\}$$

where f is primitive recursive and $f(x)$ is computed by some LOOP program. There is a (possibly very large) value x' of x such that, if $f(x) \neq 0$ for all $0 \leq x \leq x'$, then $f(x) \neq 0$ for all x (otherwise x' could be described with less than $K(x')$ bits); that is, infinite loops can in principle be transformed in finite ones. However, the value of x' is in general unknown and unknowable.

A slight modification of the arithmetic hierarchy

The Δ_n^0 class of the arithmetic hierarchy (AH) is defined in terms of the class of total recursive functions. However, this class can not be characterized by a "indexed" model of computation[2], so that there is no class of programs (or of recursive equation definition) that corresponds exactly to the class of total recursive functions. Thus, it is convenient to use a variant of the arithmetic hierarchy in which only the Δ_n^0 classes are different. We are essentially bootstrapping the arithmetic hierarchy with the class or primitive recursive predicates, instead of using, as usual, the class of total (computable) functions[3]. Define Δ_n^{PR} as the class of logical statements that can be decided with an oracle to problems in Σ_n^0; and Δ_0^{PR} as the class of primitive recursive predicates. The Kleene normal form Theorem shows that the (standard) arithmetic hierarchy can be bootstrapped with Δ_0^{PR}.

Related work

Our work differs from the important work of Hamkins and others on infinite time Turing machines [4, 2, 1] in two main aspects:

- The infinite programs considered in this work always "halt". LOOP programs always halt (without quotes). The language for infinite programs used in this work has only one more instruction than the LOOP language.
- In an infinite computation or loop, let x_1, $x_2 \ldots$ be the infinite sequence of values taken by the cell (or programming variable) x, that is, x_i is the contents of x at time t_i. In [4] when a loop finishes, the contents of the cell x is defined to be $\limsup x_i$ (which always exist), while in this work the "final" value of the programming variable x is simply $\lim x_i$, if the

[2] If ϕ_i indexes some class of recursive total functions, the function $f(n) \stackrel{\text{def}}{=} \phi_m(m) + 1$ is total recursive but not in the class.

[3] Turing, for instance, notices this essential difference in [**?**] when he says: "The class of primitive recursive functions is more restricted than the computable functions, but has the advantage that there is a process whereby one can tell of a set of equations whether it defines a primitive recursive function in the manner described above.".

limit exists[4]. If the limit does not exist, we consider the variable x to be undefined after the execution of the infinite loop; in this case, the value of x can be no more used by the program. This semantics corresponds to force a stop of every infinite loop "`for all` x..." after x takes a *sufficient large* value, that is, after all converging sequences have stabilized. There is no difference in behaviour between this semantics and the execution of the entire infinite loop. Of course, it may be impossible to know this "sufficiently large value" of x.

2 Preliminaries

The LOOP language. There are various register languages that correspond exactly to the class of primitive recursive functions. We use the LOOP language [7, 8] whose registers (or "programming variables") are x_0, x_1.... The n arguments of an n-ary function are the initial values of the registers x_1,..., x_n and the output is the contents of register x_0 when the program halts. The set of instructions is (i) $x_i \leftarrow 0$, (ii) $x_i \leftarrow x_i + 1$, (iii) $x_i \leftarrow x_j$, and (iv) `for` $x_i\{\langle$LOOP program $P\rangle\}$. In this last instruction the program P is executed a number of times that is the *initial* contents of x_i. When writing programs we will not follow this rigid syntax, using for instance other names for variables, functions known to be primitive recursive, etc.

Ordinal arithmetic.

Due to the syntactic form of iprogrs (a simple extension of LOOP programs []), every ordinal that we will use is bounded by some term of the form $a\omega^n$ where a and n are positive integers. We work with a simple form of "ordinal order of magnitude" and only need a few simple facts about ordinal number theory. For instance, for the kind of ordinals mentioned above the Cantor normal form reduces to the term with the largest exponent. More specifically, the only rules that we will use are:

[4] A similar definition was used in [**?**]

$a\omega^n + b\omega^n = (a+b)\omega^n$.

For $n > m$, $a\omega^n + b\omega^m = a\omega^n$

where a, b, m and n are integers with $a \geq 1$, $b \geq 1$.

The iprogr language

The term "iprogr" ("infinite program") denotes both a programming language and a program written in that language.

The iprogr syntax includes the instructions of the LOOP language [7, 8] characterizing primitive recursive functions, as well as infinite loops of the form

```
for all x do
    ⟨iprogr⟩
done
```

It is important to note the difference between infinite loops ("`for all` $x\{\ldots\}$") and finite, always terminating, loops ("`for` $x\{\ldots\}$"). The later is an instruction of the LOOP language, while the former is the (only) new instruction which allows the implementation of infinite programs. Infinite and finite loops will sometimes be abbreviated as "$[\ldots]$" and "$(\ldots)$" respectively.

For simplicity we do not include the "break" instruction which is used in many programming languages for aborting infinite loops. The inclusion of the break instruction would allow "faster" programs in certain circumstances. That happens namely in Σ_1^0 problems with positive answers, and in Π_1^0 problems with negative answers.

2.1 The semantics of the language

Essentially we have to define the contents of a programming variable when the execution of an infinite loop "finishes". For that purpose we will use the mathematical limit, when it exists, of the sequence associated with the variable.

In iprogrs a programming variable either contains an integer or is undefined. In an infinite loop the successive values of a programming variable define a mathematical (infinite) sequence.

Definition 1 (Undefined values) *Let* $\mathbb{N}_\nu = \mathbb{N} \cup \{\nu\}$ *where* ν *means an undefined value. Consider a sequence* x_n *defined in* $\mathbb{N}_\nu$*. We say that* $\lim_{n\to\infty} x_n = a$ *with* $a \in \mathbb{N}$*, if* $\exists m \forall n \geq m : a_n = a$*.*

Definition 2 (Sequence associated with a variable) *For every infinite loop, every programming variable* x *defines a sequence* x_n *as follows:* x_0 *is the value of* x *when the loop begins; for* $n \geq 1$*, the value of* x_n *is the contents of the programming variable* x *after* n *execution steps. The execution of each* internal *infinite loop only counts as one execution step.*

Definition 3 (Well defined programming variable) *Consider an infinite loop and let* x_n *be the sequence associated with the programming variable* x*. We say that* x *is well defined and has value* a *when* $\lim_{n\to\infty} x_n = a \in \mathbb{N}$*. Otherwise (that is, if the sequence* x_n *has infinitely many* ν*'s or is not ultimately constant), we say that,* after the execution of the infinite loop*, the value of* x *is* undefined *and write* $x = \nu$*.*

Thus, a programming variable is well defined if the value that it contains is an integer.

As an example, if x is not modified inside an infinite loop, the corresponding sequence is constant.

Definition 4 (Well defined iprogr) *An iprogr is* well defined *if for every infinite loop and every programming variable* x *with an integer initial value*

(that is, well defined at the beginning of the loop), the value of x is well defined after the execution of the loop.

Note that if the value of x is well defined at the beginning of the infinite loop, x can be undefined at the end of the loop only if x is modified (some value is assigned to x) inside the loop.

As an example, the iprogr

$$x \leftarrow 0;\ \texttt{for all}\ y\ \{x \leftarrow 1 - x\}$$

(where "$1 - x$" would be implemented with the LOOP instructions) *is not* well defined.

Definition 5 (Well defined iprogr – a relaxation of definition 4) *An iprogr is* well defined *if, after the execution of any infinite loop,* the values *of the variables that are undefined are not used*[5]*. With this definition, in any iprogr, if the input variables are well defined, the "output variable" is also well defined.*

This relaxed definition will be used in the sequel.

2.2 Restricted infinite programs

We now define a restricted form of infinite programs which is enough to "solve" all the problems of the arithmetic hierarchy.

Definition 6 (restricted iprogrs) *A* restricted iprogrs *is either a primitive recursive program (i.e. a LOOP program) or an infinite program in which every infinite loop has the form*

$c \leftarrow 0$

`for all` x *do*

[5]New values can of course be assigned to these undefined variables. Also note that, like LOOP programs, iprogrs always proceed forward, that is, there are no backward jumps (except of course, those implicit in the semantics of "`for`" and "`for all`" instructions.

⟨restricted iprogr⟩

if $y = 1$ **then** $c \leftarrow 1$

done

return c

where "$y = 1$" means: x is defined and has value 1.

The following result is easily proved.

Theorem 1 *Every restricted iprogr is well defined (Definition 4, page 7).*

In the proof of Theorem 1, page 11, the reader can see an example of a restricted iprogr that decides the statement $\forall x \exists y : f(x, y) = 0$, where f is a primitive recursive function.

Infinite programs and the arithmetic hierarchy

In general it may be difficult or impossible to determine if a limit $\lim_{n\to\infty} x_n$ exists. However, if we only use restricted iprogrs, the existence of those limits is guaranteed. Moreover, this class of iprogrs "solves" all the logical statements (Theorem 2, page 10) and some syntactic sub-classes of restricted iprogrs correspond exactly to the levels of the Arithmetic Hierarchy in the sense of Theorem 1, page 11.

The following example is an illustration of such form of infinite loops, where $S(f, x)$ is a (recursive) statement about the value of a primitive recursive function f for the argument x, such as "$f(x) = 0$":

$c \leftarrow 0$

for all x do

if $S(f, x)$ **then** $c \leftarrow 1$

done

return c

Comments The the sequence associated with c has a limit. For the case $S(f,x) \equiv [f(x)=0]$,

- the limit is 1 when $\exists x : f(x) = 0$.
- the limit is 0 when $\forall x : f(x) \neq 0$.

In the following we use the Tarski-Kuratowski Theorem ([9]) to arithmetical hierarchy.

An well defined iprogr may of course "solve" undecidable problems. In fact,

Theorem 2 *Every statement in the Arithmetic Hierarchy, or equivalently, every first order arithmetic statement is "solved" by a restricted infinite program.*

Proof. By induction. We illustrate with an iprogr that "solves" the Π_2 statement $\forall x \exists y : f(x,y) = 0$.

$$
\begin{array}{l}
c \leftarrow 1 \\
\texttt{for all } x \text{ do} \\
\quad c' \leftarrow 0 \\
\quad \texttt{for all } y \text{ do} \\
\qquad \texttt{if } f(x,y) = 0 \texttt{ then } c' \leftarrow 1 \\
\quad \texttt{done} \\
\quad \texttt{if } c' = 0 \texttt{ then } c \leftarrow 0 \\
\texttt{done} \\
\texttt{return } c
\end{array}
\tag{1}
$$

□

Based on the maximum nesting of infinite loops, we classify iprogrs in syntactic classes. This is similar to what is done for instance in [7, 8].

Definition 7 (Synctatic classification of iprogrs) *An infinite program belongs to the class $\mathcal{L}_n$ if the infinite loops are nested to a depth of at most n.*

Definition 8 *Consider an infinite loop of an iprogr and let t_n be the successive execution times of the inner part (body) of an infinite loop, not including the inner infinite loops; that is, of the inner section of LOOP code. If for every n large enough we have $t \geq a$, we say that a is an* ultimate time lower bound *("ulb") of the (infinite loop). It is assumed that all the ulb's are positive.*

Conjecture 1 (Loop nesting, arithmetic hierarchy degree, and efficiency) *For $n \geq 1$, the function can be implemented by a well defined $\mathcal{L}_n$ programs if and only if it is in the class Δ_n^{PR}. Programs in the class $\mathcal{L}_n$ have a lower bound execution time ω^n. More precisely, $t \geq (\Sigma_{1 \leq i \leq k}\, a_i)\omega^n$, where the a_i are ulb's of the infinite loops with level n.*

Note. This result is similar to Theorem 5 of [7]. Given the the execution times are infinite, it would be interesting to find a correspondence between the ordinal upper bound execution time and the class $\mathcal{L}_n$. □

Proof. Apparently not difficult. Use the arithmetic hierarchy having as degree 0 the primitive recursive predicates.

The following example will clarify the efficiency issue. Denote an infinite loop "for all $x \ldots$" by "$[\ldots]$" and the ulb's by a, $b \ldots$ The execution time t of the program

$$[\, a\, [\, b\,]\, [\, c\, [\, d\,]\,]\, e\,]\;\; f\;\; [\, [\, [\, g\,]\,]\,]$$

satisfies

$$\begin{aligned} t &\geq (a+e)\omega + (b+c)\omega^2 + d\omega^3 + f + g\omega^3 \\ &= (a+e)\omega + (b+c)\omega^2 + (d+g)\omega^3 + f \\ &= (d+g)\omega^3 \\ &= \Omega(\omega^3) \end{aligned}$$

□

3 Conclusions and future work

We mention two possible are as for further research.

3.0.1 Execution time: upper bounds

The establishment of upper (similar to the O order) or exact (similar to the Ω order) orders of magnitude for the execution if infinite programs seems more difficult.

For instance, what is the upper bound of the following infinite loop?

$$\texttt{for all}\ \ x\,\{\,\texttt{for}\,x\,\{\,\texttt{for}\,x\,\{\,\mathrm{P}\}\}\}$$

If a is a time lower bound for P, and assuming that a is constant, one would have the overall execution time

$$a + 2^2a + 3^2a + \ldots = a(1 + 2^2 + 3^2 + \ldots)$$

How do we represent this sum as an ordinal?

3.1 On program transformations

Many techniques for program transformation and optimization are known [**?**, **?**, **?**] and widely used, for instance, in compiler design [**?**, **?**, **?**]. Corresponding techniques for infinite programs have, as far as we know, not been developed.

For instance, any "classical" program with any number of (arbitrarily imbricated) loops (say "`for`" and "`while`" instructions) can be transformed in a program with a single loop, as illustrated in Figure 2, page 13.

For infinite programs such transformation is not valid. Consider iprogr (1), page 10. Here, the inner infinite loop "`for all` y" must "finish" before the next value of x is considered, and it does not seem easy (to me) to find an

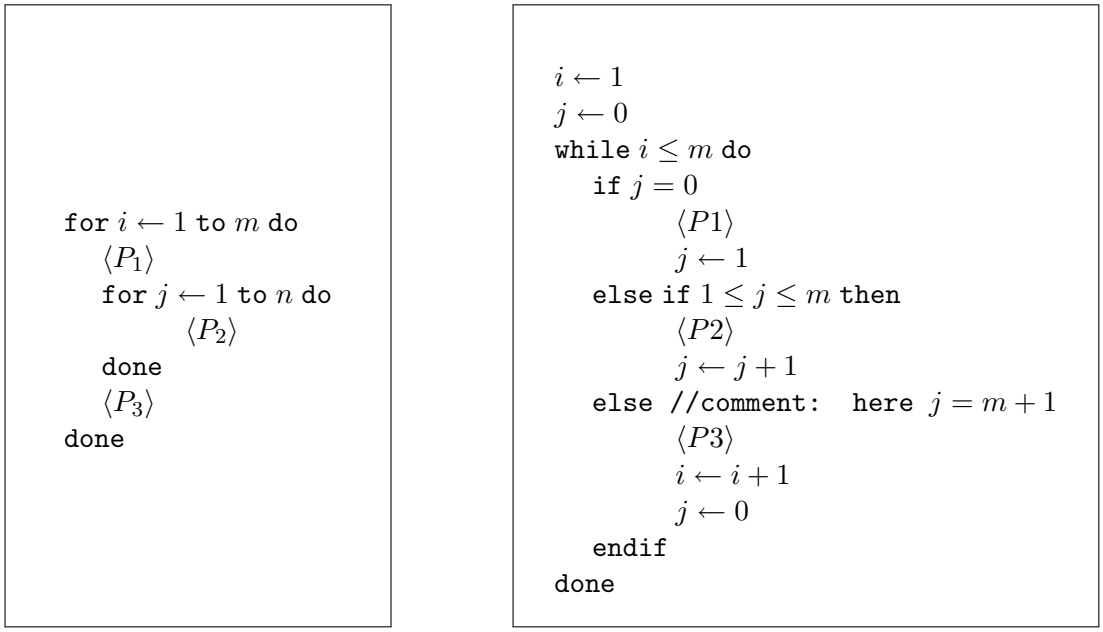

Figure 2: The program at right has a single loop and is equivalent to the program at left. For iprogrs this kind of transformation is not possible.

equivalent single (infinite) loop.

This difficulty seem consistent with the fact that we cannot express with a single infinite loop (of a restricted iprogr) an arbitrary statement of the level 2 of the arithmetic hierarchy.

References

[1] Samuel Coskey and Joel David Hamkins. Infinite time Turing machines and an application to the hierarchy of equivalence relations on the reals. url=arXiv:1101.1864v1, 2011.

[2] Joel Hamkins, Russell Miller, Daniel Seabold, and Steve Warner. Infinite time computable model theory. url=arXiv:math/0602483, 2006.

[3] Joel David Hamkins. Supertask computation. In *Classical and new paradigms of computation and their complexity hierarchies*, volume 23 of *Trends Log. Stud. Log. Libr.*, pages 141–158, Dordrecht, 2004. Kluwer Acad.

Publ. Papers of the conference "Foundations of the Formal Sciences III" held in Vienna, September 21-24, 2001.

[4] Joel David Hamkins and Andy Lewis. Infinite time Turing machines. url=arxiv.org/pdf/math/0212047, 2008.

[5] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, third edition, 2008.

[6] Benedikt Löwe. Revision sequences and computers with an infinite amount of time. *J. Log. Comput.*, 11(1):25–40, 2001.

[7] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. *Proceedings of 22nd National Conference of the ACM*, pages 465–469, 1967.

[8] A. R. Meyer and D. M. Ritchie. Computational complexity and program structure. *IBM Research Report RC 1817*, 1967.

[9] Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. MIT Press Cambridge, MA, 1987. Third printing (1992).