Theoretical Computer Science 594 (2015) 65–81

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

The efficiency of primitive recursive functions: A programmer's view

Armando B. Matos [1]

Rua da Venezuela 146 1º Dto, 4150-743 Porto, Portugal

ARTICLE INFO

Article history:
Received 25 January 2013
Received in revised form 16 March 2015
Accepted 20 April 2015
Available online 23 April 2015
Communicated by P.-A. Mellies

Keywords:
Primitive recursion
Complexity of Loop programs
Operational semantics

ABSTRACT

Using denotational semantics tools, Colson and others studied primitive recursive (p.r.) algorithms, proving the "ultimate obstinacy" property, which has as a consequence that many computations cannot be efficiently implemented in a language that faithfully expresses the classical definition of p.r. functions.

As shown by Ritchie and Meyer, the class of p.r. functions can also be characterised by the programming language Loop. Our purpose is to show that the informal, but precise, operational description of Loop (and its variants) is sufficient to prove non-trivial time lower bounds of p.r. algorithms. In particular, we present a simple proof of a property which is similar to ultimate obstinacy, namely that every p.r. program that implements a non-trivial function must execute $x_i + c$ times the body of some loop instruction, where x_i is the initial contents of some fixed input register, and c does not depend on the input values. This and other lower bounds were obtained without using the lambda calculus or systems with functional parameters (such as Gödel system T).

If the *conditional break* and the *decrement* instructions are included in the Loop language, the ultimate obstinacy property does not hold. In this case, we use another approach for obtaining lower bounds. The *unconditional break* instruction does not avoid the obstinacy property; it is equivalent (in function and efficiency) to the conditional instruction `if-then-else`. The efficient implementation of other functions like step(x) (step(0) = 0; step(x) = 1 for $x \geq 1$) and min(x, y) is also studied.

1. Introduction

The class of primitive recursive (p.r.) functions is usually defined [10,17] by a set of initial functions (the constant, successor, and projection functions) and two rules – composition and primitive recursion. It can also be characterised as the set of functions that can be implemented in a simple register programming language called Loop, see for instance [13].

Given a mathematical model or a programming language that characterises a class of functions such as p.r., we distinguish two aspects.[2]

1. *Expressiveness*: the capability of defining or programming the functions in the class (and no more).
2. *Efficiency*: the capability of defining or programming efficient algorithms for the functions in the class.

E-mail address: armandobcm@yahoo.com.
[1] Artificial Intelligence and Computer Science Laboratory (www.liacc.up.pt).
[2] Which correspond to what some authors call respectively the *extensional* and the *intentional* expressiveness.

http://dx.doi.org/10.1016/j.tcs.2015.04.022
0304-3975/

1.1. Extending the language Loop or the "classical" definition of p.r. functions

It often happens that many "formalisms", "models of computation", or programming languages, while being equally expressive (in the sense explained above), are not equivalent in terms of efficiency. Thus, in order to increase the efficiency, it may be needed to include additional instructions or to modify the recursive rules – without changing the expressiveness of the definition. Citing [16]

> ... as a practical and [...] foundational matter, we need to consider "recursive schemes" more general than primitive recursion, even if, ultimately, we are only interested in primitive recursive functions.

These "more general recursive schemes" correspond, in the Loop language, to new program constructs. The examples of the Loop extensions studied in this paper are:

- The unconditional "`break`" instruction (which immediately breaks a loop) that allows the implementation in constant time of the conditional `if-then-else` instruction. In fact, the two instructions, as well as the `if-then` instruction are "efficiently equivalent".
- With the conditional "`break if zero`" instruction (which breaks a loop if the contents of a register is 0) plus the "`decrement`" instruction, it is possible to implement the $\min(x, y)$ function in time $O(\min(x, y))$, thus avoiding a manifestation of the obstinacy problem, see for instance [3,4,16]. The "`break if zero`" and the "`break if not zero`" instructions are equivalent in terms of efficiency. They are also both related to the step(x) function (see definition in page 67).

The Loop language and its extensions will be referred as "Loop languages".

1.2. Purpose of the paper

The main question this paper addresses is whether the lower bounds for the complexity of p.r. algorithms, such as those mentioned in [3,4,16,18], can be directly obtained. By "directly" we mean using only an informal, but precise, description of the Loop languages, and without using the lambda calculus or systems with functional parameters (such as Gödel system T). The answer is yes. Using this "direct approach", we prove, for instance, the "obstinacy-like" results of the seminal work [3].

1.3. Look ahead

The reader may find it interesting to look at

- Fig. 2, page 80.
- Fig. 3, page 80.

where he can find a summary of the results of this paper.

1.4. Related work

Several authors, see for instance [3,4,16,15,18], studied the efficiency of primitive recursive definitions, using the lambda calculus to express the denotational semantic of programming languages, and adaptations of Gödel's system T to describe specific variants of the Loop language.

The lambda calculus has been used by [6] to express the operational semantics of Loop. Corollary 13 of [6] (a consequence of Theorem 12) corresponds to Theorems 1 (page 73) and to Corollary 2 (page 76); see also Sections 5.3.1 and 5.3.2.

Modifications of the basic recursion schema and several extensions of the Loop language have also been studied. For instance (i) the instruction "`dec`x" is often considered as part of the Loop language, see for example Andary et al. [1], Michel and Valarcher [15]; (ii) the conditional construct is studied by several authors, see for instance [18]; (iii) first class procedures and function pointers, "the imperative counterpart of Gödel system T", are considered in [7,18]; (iv) the "`exit`" instruction is used in [1]; (v) a conditional form of "exit" and a conditional loop instruction[3] are used in [15].

1.5. Paper organisation

The paper is organised as follows. Section 2 (page 67) contains the background needed for the rest of the paper. The "Loop languages" are described in Section 3 (page 67). Several lower bounds associated with the algorithms implemented in those languages are proved in Section 4 (page 72). The implementation of several programming constructs and of algorithms

[3] In terms of the current paper this instruction is similar to the `for` loop together the `breakZ` instruction.