Descriptive Complexity of a Fragment of C Language

Soha Safwat
Faculty of Computer Science
October University for Modern Science and Arts
6th October, Egypt
Soha.cs@gmail.com

Wafik Lotfalalah
Faculty of Engineering
AUC University
Cairo, Egypt
lotfalla@uwalumni.com

Ibrahim Farag
Faculty of Computers and Information
Cairo University
Giza, Egypt
I.Farag@fci-cu.edu.eg

Abstract—Descriptive complexity emerged from finite model theory and had its aims in finding logics that capture computational complexity classes. Observing that imperative programming languages can also be considered as logics (maybe with some additional control), we extend the scope of descriptive complexity by defining two fragments of such languages and showing that they capture the classes P and NP.

Keywords—Descriptive complexity; Language A; Fragment for P

I. INTRODUCTION

Descriptive Complexity was initiated by Ronald Fagin in his doctoral thesis [3], where he showed that the class NP is characterized by existential second order logic SO².

Since then, many important complexity classes were characterized by their corresponding logics. For example, the complexity classes L, NL, P, and PSPACE were shown to be equivalent to first order logic augmented with the deterministic transitive closure, the transitive closure, the least fixed point, and the partial fixed point operators, respectively. See [2] for a nice exposition of these (and other) results.

However, for most CS students, such characterizing logics are foreign languages. In fact, the field of descriptive complexity would have been much more successful, had it been speaking the same languages CS scholars understand, namely C, Java, or other imperative languages. We should note here that these languages are themselves logics in their own right, maybe with some additional control that defines how programs are executed on a machine.

Thus, we are lead to the following natural question: Can we find fragments of imperative programming languages that capture important complexity classes? The main goal of this paper is to answer the above question affirmatively. We give some nice (and easy) characterizations of the class P using syntactical

We start by outlining a general recipe for finding such characterization: To show that some syntactical fragment A' of A captures a complexity class C, we follow the following plan:

1. We define the syntax of A' by restricting the syntax rules of A.
2. We show that problems decided by programs of A' must be in C.
3. We write a program in A' that solves a complete problem for C via first order reductions.
4. We show that A' is closed under first order reduction. In other words if \( I \) is a first order query which reduces a problem \( P \) to a problem \( Q \), i.e. \( x \in P \) if and only if \( I(x) \in Q \), and \( Q \) is decided by an A' program, then \( P \) is also decided by an A' program. This shows that \( C \subseteq A' \), and thus \( A' = C \).

This paper is organized as follows: We start in Section 2 by giving the definition of the decision Language A, which is equally powerful to C (or any other programming language). In Section 3, we define the syntactical fragment A₀ and show that it captures the class P. Section 4 concludes the paper by some pointers to future research.

II. THE LANGUAGE A

We define in this section an abstract C-like language called A. This language has just two basic data types, namely Boolean and (non-negative) Integer. Complex data types can be recursively formed of one dimensional array of simpler types. A Boolean variable takes the storage of just one bit, while an Integer variable is considered (for definiteness) to take the storage of 64 bits, thus it only holds a natural number between 0 and 2⁶⁴ - 1.

Annual International Conference on Infocomm Technologies in Competitive Strategies (ICT 2010).
Copyright © GSTF 2010.
doi:10.5176/978-981-08-7240-3_i-19
For example, a (directed) graph can be represented by a pair $(n;E)$, where $n$ is of type Integer and denotes that the graph vertices are numbered $1;...; n$. $E$ is an array $E(1) ;...;E(n)$, where each $E(i)$ is an array $E(i)(1) ;...;E(i)(n)$, where $E(i)(j) = E(i; j)$ is a Boolean variable which is true if and only if there is an edge connecting $i$ to $j$. Thus $E$ is the $n \times n$ incidence matrix for the graph. Programs in this language are decision connecting variable which is true if and only if there is an edge $E$ «$Q$ « $x$ and return a Boolean value (= the value of $f$). Such functions are called $A$-functions. These start with a header of the form $f(x_1, ..., x_k)$, followed by a section, where local variables are declared, followed by the body of $f$, which is a list of instructions of the following types:

- **Simple Assignment Statements:** ($y := f(x_1, ..., x_k)$) The left hand side must not be $n$, the recursive assignment statement must not occur within any loop, and $x_1$ must be $n/2$.
- **Recursive Assignment Statements:** ($y := f(x_1, ..., x_k)$) The left hand side must not be $n$, the recursive assignment statement must not occur within any loop, and $x_1$ must be $n/2$.
- **Conditional statements:** No restriction.
- **For Loops:** (For $i = 1$ to $n$ do $P$) $P$ must not modify the value of $i$. Here the $n$ used must be the nesting variable $n$ at this state.
- **While Loops:** (While $C$ do $P$) Such loops must be preceded by an assignment statement $i := 1$ (where $i$ denotes a counter), $C$ must have the form $(i <= 0$ and $C_0)$, $P$ must have the form $P_0$; $i := i + 1$, and $P_0$ must not modify the value of $i$.
- **Function call:** (Call $f(x_1, ..., x_k)$) The function call must not occur within any loop, and $x_1$ must be $n/2$.
- **Termination:** No restriction.

### A. Language Overview

#### 1) Data Types

C has a wide variety of data types starting from numerical data types which vary in size and precision, going through the Boolean data type, the char and the string. These types can be shown below:

- byte: 8 bit integer 2's complement
- short: 16 bit integer 2's complement
- int: 32 bit integer 2's complement
- long: 64 bit integer 2's complement
- float: 32 bit single precision floating point
- double: 64 bit double precision floating point
- char: 16 bit space that carries one Unicode character
- boolean: 1 bit space that can only carry a value of "true" or "false"

There is also the String which is an unlimited space that can carry an unlimited number of characters. Although it is not a basic data type like the others, the programmer can treat it like one because C takes the trouble of handling it.

As you can see the above types are more than enough to solve most of the problems and all the basic operations used on those types (except for the string) are done in constant time but for those who want to go to the extremes and use more complicated types like big integer, object orientation is always available in order to help you create your own data type class and your own operations to use on that data type.
2) **Literals**

C has different types of literals:

1. **Number Literals**: like integer literals that can have octal, decimal or hexadecimal bases and floating point literals with standard and scientific notations.

2. **Boolean Literals**: the 2 words "true" and "false" which have only a logical meaning with no numerical representation.

3. **Character and String Literals**: unicode characters represented in single quotes for a single character and in double quotes for a string. There is also the escape sequences used to express special characters (like n’ for a single quote).

3) **Operators & Expressions**

C benefits from a wide a variety of operators with different types. There are the assignment (=), arithmetic (+,-,* ,/ ,% ), logical ( && , || , ! ), increment (++, -- ), relational ( < , > , <= , >= , == , != ) and bitwise and shift operators ( | , ^ , & , >> , << , >>> ). Each of these operators has its own precedence over others.

4) **Variable Assignment**

Variable assignment is quite easy in C. There are three basic types of assignment:

- **Simple Assignment**: where a variable is assigned a direct value (e.g. int x=5; or char y = ’d’;) or the value of the result of a basic operation (built in arithmetic and logical operations e.g. Boolean h = a && b; or int x = 200+7 ;).

- **Functional Assignment**: where a variable is assigned to the value returned by a method on the condition that the return type of the method matches the data type of the variable (e.g. int x = A( ); where A is a method that returns int).

- **Type Casting**: used to convert one data type into another under certain conditions. It is usually used in the input where the user enters characters from the keyboard which are automatically converted to a string then this string can be type casted into an int if it has only numeric characters (e.g. int x = (Integer) word; where word is a string containing only numeric characters). It can also be used to convert objects from type object to more specific types.

5) **Control Statements**

- If else statements (if condition then do S1 else do S2) (where S is a statement)
- Switch statements (switch char or int x, case x = value, do S)

6) **Iterative Statements**

- while loops: while condition do S
- do while loops: do S while condition
- for loops: for i = 0 to n do S

7) **Functions**

C is very flexible yet powerful when it comes to functions. Each function has a unique signature that consists of the return type of that function, its name and its parameter list. Any change in one of these things can yield a different function. They can be classified into different types depending on different function properties. We can have static and non static functions, void and returning functions and recursive and non recursive functions. However, for the sake of simplicity, we will classify them in our own way in order to help us achieve our goal. Our classification is based on how different types of functions call each other as follows:

![Fig. 1 Functions are classified based on how they call each other.](image-url)

All functions will be classified into 3 main groups that are shown in Figure 1.1. A function is represented by a node on the graph. Function A directly calls function B if there is an arrow going from A to B, while function A indirectly calls function B if there is a path from A to B that has more than one edge. The three function types are:
1. Type 1 functions: non recursive functions that do not directly or indirectly call other recursive functions (e.g. functions f & g).
2. Type 2 functions: directly or indirectly recursive functions. Functions that call themselves by themselves or by calling other functions. In other words any function that belongs to a cycle on the graph (e.g. b, c, d, e & h). These functions can call Type 1 functions.
3. Type 3 functions: These are functions that are neither Type 1 nor Type 2 but can directly or indirectly call either type freely (e.g. a & i).

8) Data Structures
C has got a wonderful group of different data structures that will fit almost all the needs to solve different kinds of problems. These structures are generic classes that store objects and come with different class methods to deal with them. They come in a variety of types like Arrays, Linked Lists, Vectors, Hash Sets, Priority Queues and much more.

9) Objects and classes
It is very easy to create a class in C. Once you have defined your class variables, constructor and other class methods, you can easily create a new object using the "new" keyword. C also supports different aspects of object orientation like Abstraction, Inheritance, Encapsulation and Polymorphism which makes it easy for a programmer to solve the problem in an object oriented manner.

- \{X\} denotes 0 or more occurrences of X
- A | B denotes the occurrence of A or B
- A || B denoted the occurrence of A or B or both

N.B: This grammar is only for representing the structure of the language and not the syntax.

B. Theorem $A_p = P$.

1. A call of an $A_p$ function $f(n; x_2, ..., x_k)$ takes time that is polynomial in $n$.

Thus $A_p \subseteq P$.
2. There is an $A_p$ function that solves the circuit value problem CVP. Given an acyclic Boolean circuit with specified inputs, find its output. Note that CVP is complete for P via first order reductions [Imm83].
3. $A_p$ is closed under first order reductions.

Proof Outline:

(1) Is done by induction on $A_p$ functions.
(2) We show how an $A_p$ function can solve the circuit value problem (CVP). An $A_p$ function for CVP, will first take the size $n$ of the circuit (= the number of gates it has). For simplicity assume that the gates are ordered such that each gate takes its inputs from the outputs of previous gates only. The function then uses two nested for loops. The outer loop cycles over all gates in order, evaluating their outputs, where at each gate, the inner loop checks the outputs of all previous gates connected to it and decides the output of this gate according to whether it is an “and”, “or”, or “not” gate.

It is clear that such a program satisfies all restrictions for the $A_p$ fragment. Finally, we need to show (3) i.e. $A_p$ is closed under first order reductions. In other words, if I is a first order query mapping an input string $x$ to an input string $y = I(x)$, and $f(y)$ is a Boolean function evaluated by an $A_p$ -function, then the Boolean function $g(x) = f(I(x))$ can also be evaluated by an $A_p$ -function.

The code for the function $g(x)$ is essentially that of $f(y)$, but every time the code mentions an input variable $y$ in $y$, the query $I$ is invoked on $x$ to decide about the required value of $y$.

There are two points, however, that we need to take care of:

- The nesting parameter $n$ (first variable of $x$) may be larger than the nesting parameter $m$ used in $f(y)$. However, since the size of $y = I(x)$ is polynomial in the size of $x$, loops in $f(y)$ may be replaced by few nesting of loops in $g(x)$.
- Since the first order query involves just Boolean operations together with universal and existential quantifiers ranging over $x$,
such queries can easily be coded in $A_p$ (with possibly several nested loops for each quantifier to scan the whole size of $x$ that is polynomial in $n$).

IV. RULES TO CAPTURE $P$

1. All the data types, literals, operators and expressions described above can be used freely with no restriction because operating on them requires constant time due to the presence of specialized dedicated hardware for each of those operators.

2. Size of the input must be expressed in the form of a variable $n$ that has the following properties (Assuming $|\text{Input}| = m$):
   - $n$ must be polynomial in $m$ ($n = O(m^k)$) where $k$ is a constant.
   - $m$ must be polynomial in $n$ ($m = O(n^j)$) where $j$ is a constant.

This $n$ will always be our measure for complexity and it will work quite well since $P$ is closed under composition. In this way any expression that is polynomial in $n$ would also be polynomial in $m$ and vice versa.

3. All variable assignment methods can be used on the condition that they don't change $n$ throughout the whole program. Once $n$ takes a value it is not allowed to be changed because it is an indication for the input size which is fixed throughout the program. Any change in $n$ would affect the loop counters and may cause the loops to exceed polynomial time.

4. All conditional statements like If else and switch statements can be used without restriction because checking the condition in most cases will take constant time if it uses basic operators (e.g. $<, \&\&$, $k$) in this case the condition itself doesn't affect the time but rather the number of checks done which will be dealt with in the iterative part. In other cases the condition may depend on the return value of a Boolean function which may take more than constant time but that also won't make a problem assuming that all the methods used will take polynomial time to finish.

5. Iteration can only be performed using for loops which declare and initialize the counter in the for loop statement because that is the only way to ensure that the counter will not be changed outside the loop. If the counter was declared outside the loop, like the case with other loop statements, we can't guarantee that it will not change, because even if we check that it is untouched within the loop body, the loop can make method calls which can change it. In addition to that a for loop is such as powerful as any other loop.

6. Counters in a for loop must never be changed in the body of the loop but rather incremented in the initial for loop statement.

7. Loop termination limits can only be either constant or polynomial in $n$ as this is the only way to ensure that the loop won't exceed polynomial time. The difference between the initialization and termination limits of the counter must be polynomial. This has the benefit of focusing on the number of iterations executed by the loop which we are interested in rather than the specific value of the counter. However, to make it easier to check, we will assume that all counters will start from zero. Of course it is needless to say that infinite loops are not accepted.

8. Branching statement like "break" and "continue" can be used on the condition that they are not labeled because if we use a labeled break, we can jump to unexpected different parts of the program causing it to go in an infinite loop. Exception handling statements (e.g. try, catch, and throw) can also be used freely.

9. Type 1 function can contain loops and can also be called from inside loops.

10. Type 2 functions must take an integer parameter called limit and they must return whenever it reaches zero. This variable will act as a limit on the number of levels in the recursive tree.

11. Recursive calls of Type 2 functions must occur with a level limit = (limit/k) where $k > 1$. If they have a branching factor $b > 1$, this limit must be decreased dynamically by a factor of $k$ during the recursion process. This will ensure that the recursion tree would have a logarithmic number of levels and a polynomial number of nodes. However, if $b = 1$ recursive calls can take limit-1 as a level limit as they will behave as a normal loops in that case.

12. Number of recursive calls inside a recursive function (directly or indirectly recursive) must be constant. Functions calling themselves by themselves or by other functions can't do the calling in a loop that has counter dependent on $n$. However, these functions can contain $n$ dependent loops on one condition that they do not make recursive calls within these loops because now each of these loops will be executed each time the method is called; in other words, they will be executed $c$ times; where $c$ is the total
number of nodes in the recursive tree. Therefore, the total number of steps executed in all the loops along the recursive tree = n*c which is still polynomial in the size of the input because c is polynomial.

13. Type 3 functions can call all types of functions from inside or outside a loop. Recursive functions must be called from these functions with limit =n

14. Object Oriented features and the new keyword can be used freely as they will not affect the running time of the program.

V. CONCLUSIONS

The field of descriptive complexity has successfully linked classical logics to CS, and revealed the machine-independent characterizations of complexity classes. However, it was not very successful in penetrating the computer science undergraduate curricula. This paper is trying to change this state of affairs, by linking descriptive complexity to imperative programming languages, the CS students are most familiar with. We defined a fragment of a C-like programming language, and showed that they capture exactly the class P. Such characterization was not hard, as most of the theoretical work has been already done in Finite Model Theory. The results here can easily be modified to capture EXPTIME, L, NL, and PSPACE. However, for randomized complexity classes, things are not clear.

REFERENCES
