Bounded Complexity Languages

Martin Kolář

Abstract

This theoretical paradigm allows the deduction of complexity directly from code, rather than by analysis of all possible operations.

1 Introduction

Important problems in theoretical computer science, such as P vs NP, have been formulated and analysed with complexity bounds defined on a Turing Machine. However, a Turing Machine can execute algorithms of arbitrary upper bound complexity, and the complexity of arbitrary programs cannot be predicted in general. This is known as the Halting Problem.

This paper presents Bounded Complexity Languages, in which every program halts. A constructive proof shows that any algorithm with a given upper bound in a Turing Complete system can be transcompiled into a Bounded Complexity Language of the same upper bound complexity.

Theorem 1.1 *Any program with an upper bound complexity on a Turing Machine can be translated into a sequential program with counted loops whose iterations are bounded by the same complexity.*

Rather than the standard approach, which defines upper bound complexity by the highest number of operations that the entire algorithm performs for any input. Thus, any algorithm which cannot be written in a Bounded Complexity Language of a given bound is guaranteed to have a higher bound.

2 Proof Outline

A sequential programming language is constructed in such a way that it allows only counted loops, and no recursion. A procedure demonstrates that any program with a polynomial upper bound complexity can be transcompiled into a Polynomial Bounded Complexity Language program. The construction is shown with a polynomial upper bound, but any other function can be used (logarithmic, linear, exponential, ...).

3 Language Definition

Define polynomial programming language as a sequential programming language with variables, functions, and operations, such as C. However, the following restrictions apply:

A no *while* loops

B functions are uniquely numbered $\mathbb{N}$

C any function f_a is only allowed to call lower-numbered functions f_b where $a > b$

D the number of iterations of *for* loops is bounded by a polynomial of order q given the input problem size n

These properties assure that there is no recursion, and that any program written in such a language has an upper bound $O(n^q)$

Corollary 3.0.1 *Every polynomial time algorithm on a Turing Machine can be written in a polynomial language.*

And therefore:

Corollary 3.0.2 *Every program of a polynomial programming language halts.*

4 Constructive Proof

This procedure converts any polynomial program in any Turing Complete system into a program of a Polynomial Bounded Complexity Language, without executing the program.

1 Convert into a sequential programming language with *while* loops and function calls. This is possible by the definition of Turing Completeness.

2 Map function calls in a graph G

3 Identify loops in graph G, copy the code, and convert each loop into a *while* loop in a new function. Repeat this step until all loops in G are eliminated. The simplest case is a function calling itself. This is done for all possible circular function calls, without analysing whether they are called an runtime.

4 Uniquely number all functions such that a function f_a calls only functions f_b such that $a > b$

5 Convert all *while* loops with condition C into *for* loops which iterate n^q times, and *break* when the condition C is satisfied.

Since the resulting language is Turing Complete, any resulting program can be transcompiled into a Turing Machine system.