Expressing Computational Complexity in Constructive Type Theory*

Robert L. Constable
Cornell University

Abstract

It is notoriously hard to express computational complexity properties of programs in programming logics based on a semantics which respects extensional function equality. This is a serious impediment to certain key applications of programming logics, even those which apply very well otherwise.

This paper shows how to define computational complexity measures in such logics as long as they support inductively defined types, dependent products, and functions. The method exploits a natural feature of inductive definitions in type theory, namely that implicit codes are kept with the objects showing how they are presented in the inductive class.

The adequacy of the proposed definition depends on a faithfulness theorem showing that the external (or meta-level) definition of complexity is respected by the internal definition. The results are applied to defining resource bounded quantifiers that can be used to state complexity constraints on constructive proofs and their extracted programs. In such resource bounded logics it is possible to prove theorems like a PTime axiom of choice. The results of the paper bridge the fields of semantics and complexity to a small extent.

1 Introduction

Most programming logics use this rule for function equality

\[ f =_{A \rightarrow B} g \iff \forall x : A. f(x) =_{B} g(x). \]

The functions \( f \) and \( g \) may be given by programs, say that \( f \) and \( g \) are also names for the programs. In the meta-theory of the programming logic, we have access to a finer equality, the equality on \( f \) and \( g \) as programs or terms. Given

*Work supported in part by NSF grant CCR-9244739 and ONR grant N00014-92-J-1764
this access to the program structure, we can define the usual computational complexity measures, say \( \text{time}(f) \) and \( \text{space}(f) \). But in the object theory, we lose access to these functions since they do not respect the function equality.

One approach to gain access to computational complexity in the object logic is to define a finer equality on functions, say some intensional equality \([10]\). But experience has shown that such logics are difficult to use and to interface with conventional mathematics. Another approach is to use a logic, say Bounded Linear Logic \([16]\), that keeps track of computational resources. This approach also requires a great deal of as yet unfinished work to show that such an axiomatization of programming is manageable.

The issue in this paper is to look for an existing mechanism in a constructive programming logic that can be exploited to define computational complexity in the object logic in a natural way.

The basic idea is to notice that the computational interpretation of an inductively defined class of functions, say \( C(A \rightarrow B) \) defined over \( A \rightarrow B \), contains in it an implicit system of codes that can be used to define complexity. We will show that given \( f \in C(A \rightarrow B) \) we can define \( \text{time}(f) \) and \( \text{space}(f) \) in terms of these codes. This is possible because equality on inductive classes is not the same as equality in the underlying type. We are exploiting a fact that is naturally part of constructive inductive definitions, not introducing a new mechanism just for the sake of defining complexity. This mechanism is also present in programming languages like ML that support inductive types.

2 Basic Concepts

The results are given for an especially simple but powerful type theory. Its base types are just \( \text{unit} \), \( 1 \), and \( \text{void} \) (true and false under propositions-as-types) and the types \( \text{Type}_i \). The element of \( \text{unit} \) is just a dot, \( . \). The constructors are dependent functions (\( \Pi \)) and dependent products (\( \Sigma \)). We denote the language by \( \Pi \mu \Sigma^+ \) (a Greek acronym for “Promise”).

\[
\Pi \bar{x} : A. B(x) \quad \text{are those functions } \lambda(x, b) \text{ such that for all } \\
a \in A, \; f(a) \in B(a) \quad (f(a) \text{ is also written } ap(f, a)).
\]

\[
\Sigma \bar{x} : A. B(x) \quad \text{are those pairs } <a, b> \text{ such that } \\
a \in A \text{ and } b \in B(a).
\]

It must be that \( A \) is a type and \( B(a) \) is a type for each \( a \in A \). We also allow the disjoint union, \( A + B \), of types \( A \) and \( B \). The elements are the injections, \( \text{inl}(a) \) and \( \text{inr}(b) \).

Finally, \( \mu(X. F) \) is a recursive type, in \( \text{Type}_i \), provided \( F \) is a monotone function \( \text{Type}_i \) to \( \text{Type}_i \). We say that \( a \in \mu(X. F) \) iff \( a \in F[\mu(X. F)/X] \). With each