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

$$f =_{A \to B} g \text{ iff } \forall x : A.\ f(x) =_B g(x).$$

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 $time(f)$ and $space(f)$. 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 $\mathcal{C}(A \rightarrow B)$ 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 $f \in \mathcal{C}(A \rightarrow B)$ we can define $time(f)$ and $space(f)$ 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 *unit*, $\mathbf{1}$, and *void* (true and false under propositions-as-types) and the types $Type_i$. The element of *unit* is just a dot, $\cdot$. The constructors are *dependent* functions (Π) and dependent products (Σ). We denote the language by $\Pi\mu\Sigma^+$ (a Greek acronym for "Promise").

$\Pi x : A.\ B(x)$ are those functions $\lambda(x.\ b)$ such that for all $a \in A,\ f(a) \in B(a)$ ($f(a)$ is also written $ap(f,\ a)$).

$\Sigma x : A.\ B(x)$ are those pairs $< a,\ b >$ such that $a \in A$ 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*, $inl(a)$ and $inr(b)$.

Finally, $\mu(X.\ F)$ is a recursive type, in $Type_i$, provided F is a monotone function $Type_i$ to $Type_i$. We say that $a \in \mu(X.\ F)$ iff $a \in F[\mu(X.\ F)/X]$. With each

recursive type $\mu(X.\ F)$ there is a recursion combinator, μ–*ind*, which defines the $\mu(X.\ F)$ –recursive functions. The typing rule is

$$\frac{H \vdash a \in \mu(X.\ F) \quad X : Type_i,\ z : F,\ f : X \rightarrow G \quad \vdash g \in G}{H \vdash \mu\text{–}ind(a; z,\ f.\ g) \in G}$$

examples

example 1: The natural numbers can be defined as $\mu(N.\ \mathbf{1} + N)$ with $0 = inl(\cdot)$ and $succ(x) = inr(x)$. The primitive recursive functions are defined by the combinator μ–*ind* (details are not critical to the results).

example 2: We define the set of functions $N \rightarrow N$ built from given base functions B by composition. Suppose the composition operator is $C(f,\ g) = \lambda(x.\ f(g(x)))$. Intuitively we are defining $\mathcal{C}(N \rightarrow N) = B + Comp(\mathcal{C}(N \rightarrow N),\ \mathcal{C}(N \rightarrow N))$ where

$$Comp(S_1,\ S_2) = \{f : N \rightarrow N \mid \exists g_1 : S_1,\ \exists g_2 : S_2.\ f = C(fun(g_1), fun(g_2))\}.$$

The official definition of $\mathcal{C}(N \rightarrow N)$ is $\mu(F.\ B + Comp(F,\ F))$. Notice that if b_i are base functions, then an object like $inr(\mathcal{C}(inr(\mathcal{C}(inl(b_1),\ inl(b_2))),\ inl(b_1))$ is an element of $\mathcal{C}(N \rightarrow N)$. Given $f \in \mathcal{C}(N \rightarrow N)$, we need the mapping

$$fun : \mathcal{C}(N \rightarrow N) \rightarrow (N \rightarrow N)$$

which "pulls out" the function part of the object, e.g.

$$fun(inr(C(inr(C(inl(b_1),\ inl(b_2))),\ inl(b_1))) = C(C(b_1,\ b_2),\ b_1)$$

There is another part of the object hidden in the definition, namely the sequence of injections *inl*, *inr* and the access to B. So we can define $code(f) = inr(inr(inl(1), inl(2)),\ inl(1))$. This code tells us the structure of the definition of $fun(f)$. Based on the code we can measure the complexity of this particular *presentation* of f based on using a specific means of computing a function in $N \rightarrow N$ that is guaranteed to be equal to f.

3 Defining Computational Complexity

3.1 Meta-level complexity

A careful definition of the language, $\Pi\mu\Sigma^+$ (Promise), would be based on a class of terms. The definition would include these clauses. (It can define most of Nuprl [8].)

$$\frac{x \in var}{x \in term} \qquad \frac{t \in term}{\begin{array}{c} inl(t) \in term \\ inr(t) \in term \end{array}} \qquad \frac{x \in var,\ t \in term}{\lambda(x.\ t) \in term} \qquad \frac{a \in term\ f \in term}{ap(f,\ a) \in term}$$

$$\frac{a \in term,\ b \in term}{pair(a,\ b) \in term} \qquad \frac{x \in var\ t \in term}{\mu(x.\ t) \in term} \qquad \frac{t \in term,\ z \in var\ f \in term\ g \in term}{\mu{-}ind(t; z,\ f.\ g) \in term}$$

Computation is defined by structured operational semantics, for example, we have these rules among others.

$$\frac{f \downarrow \lambda(x.\ b) \quad b[a/x] \downarrow c}{ap(f,\ a) \downarrow c} \qquad \frac{g[a/z,\ \lambda(x.\ \mu{-}ind(x; z,\ f.g)/f] \downarrow c}{\mu{-}ind(a; z,\ f.\ g) \downarrow c}$$

We can define computational complexity based on this kind of scheme or on a rewrite sematnics. For example, a simple step count can be defined by

$$\frac{time(f) \downarrow n \quad time(b[a/x]) \downarrow m}{time(ap(f,\ a)) \downarrow (n + m + 1).}$$

This approach allows us to define both an evaluation function and a time-complexity function on terms. Given f a function term

$$\begin{array}{ll} eval(f) & : \{x : term \mid \exists y : termf(x) \downarrow y\} \rightarrow term \\ time(f) & : \{x : term \mid \exists y : term.\ f(x) \downarrow y\} \rightarrow \mathbb{N}. \end{array}$$

These are the *metalevel* evaluation and time-complexity functions. In some sense they give the actual complexity of terms relative to a particular implementation of the programming language.

3.2 Reflected computation

In a language with recursive types and dependent types like $\Pi\mu\Sigma^+$ it is easy to reflect the term structure and evaluation structure into the object language, building the internal type Term and the internal Eval and Time functions. This was done in detail for Nuprl [1]. We will briefly refer to these ideas later.

3.3 Computational complexity

summary of problem and a solution

The technical barrier between semantic theories and complexity theory is that semantics deals with (computable) *functions* in order to interface with conventional mathematics, and complexity theory deals with *algorithms* since costs depend on the details of the algorithm. This distinction comes down mainly to this rule for function equality used in most programming logics.

$$f =_{A \to B} g \text{ iff } \forall x : A.\ f(x) =_B g(x).$$

inductive classes of functions

Let's look at the form of inductive definitions in type theory [8, 9, 26, 14]. The ideas are similar to those in Feferman and the methods are like those of Scott [27]. If F is a monotone operation on types, say $X \subseteq Y \Rightarrow F(X) \subseteq F(Y)$, then $\mu(X.\ F)$ is a type that is essentially the least fixed point of F. This fact is characterized by equipping $\mu(X.F)$ with an induction principle. Informal notations often look like $X \stackrel{def}{=} F(X)$. So we might write $N \stackrel{def}{=} \mathbf{1} + N$ for $\mu(N.\mathbf{1} + N)$.

We are going to define a class of functions that has the form $\mu(F.\ \mathsf{Base}+\mathsf{Rec}(F)+\mathsf{Comp}(F))$ where Base is a collection of base functions and Rec and Comp define classes based on a recursion combinator and composition. We will pick Base, Rec, and Comp so that the classes defined are the polynomial time functions. The key to doing this elegantly is in the work of Leivant [22] from LICS '91; the particular result I use (at Leivant's suggestion) is from Bellantoni and Cook [3].

Let $N = \{0, 1\}$ *list*. This represents the natural numbers in the usual way (with degenerate leading 0, low order bits at the head). Let $N^0 = \mathbf{1}$, the unit type, and $N^{n+1} = N \times N^n$.

The functions we study have two kinds of numerical inputs, called *normal* and *safe* in [3]. We think of them as *canonical* inputs (or normal) and *non-canonical* (or non-normal). So they have type $N^n \times N^m \to N$. The left arguments are canonical (N^n). We use N^0 to indicate the absence of an input. The various function spaces are collected into a single domain, $\mathbb{D}$, by taking the disjoint union.

Def. $\mathbb{D} = \sum n : N.\ \sum m : N.\ \ N^n \times N^m \to N$.

The form of any element of $\mathbb{D}$ is $\langle n, m, \boldsymbol{f} \rangle$ for $\boldsymbol{f}$ the function. The selection functions for $f \in \mathbb{D}$ is $norm(f) = n$, $safe(f) = m$, $fun(f) = \boldsymbol{f}$.

Let $\{x : A \parallel B_x\}$ abbreviate the dependent product type, $\Sigma x : A.\ B_x$. The elements are $\langle a,\ b \rangle$ with $a \in A$ and $b \in B_a$. We also use the Nuprl phrase $\downarrow(P(x))$ for P a proposition. This is a proposition whose computational content

has been "squashed." The official definition is $\{unit \mid P(x)\}$, so the value is $\cdot$ if $P(x)$ is true, otherwise the type is empty.

The Base functions are divided into five classes $C_0, \ldots, C_4$.

Def.

$$\begin{aligned}
C_0 = \quad & \{f : D \mid \quad norm(f) = 0 \\
& \qquad \& \ safe(f) = 0 \\
& \qquad \& \ fun(f) = \lambda p.\ 0\} \\
C_1 = \quad & \{f : D \mid \quad \exists n, m, i : N.\ \ norm(f) = n \\
& \qquad \& \ safe(f) = m \\
& \qquad \& \ fun(f) = proj(n, m, i) \\
& \qquad \& \text{ if } n = 0 \text{ then } i \neq 1 \\
& \qquad \& \text{ if } m = 0 \text{ then } i \neq n + 1\}
\end{aligned}$$

These are the projection functions where $proj(i, n, m)(x_1, \ldots, x_n; x_{n+1}, \ldots, x_{n+m}) = x_i$ provided we don't project out the element of $\mathbf{1}$.

$C_2 = \{f : \mathbb{D} \mid \exists j : [0, 1].\ norm(f) = 0 \ \& \ safe(f) = 1 \ \& \ fun(f) = s_j\}$. These are the two successor functions, $s_0(\cdot; x) = 0x \quad \text{and} \quad s_1(\cdot; x) = 1x$.

$C_3 = \{f : \mathbb{D} \mid norm(f) = 0 \ \& \ safe(f) = 1 \ \& \ fun(f) = pred\}$. This is the predecessor function, $pred(\cdot; 0) = 0$ and $pred(\cdot; ix) = x$.

$C_4 = \{f : \mathbb{D} \mid norm(f) = 0 \ \& \ safe(f) = 3 \ \& \ fun(f) = cond\}$.

This is the conditional function where

$$cond(\cdot; a, b, c) = \ \ if\ a\ mod\ 2 = 0\ then\ b\ else\ c\ fi.$$

Notice that we separate the canonical arguments from the non-canonical with a semicolon.

Along with these types, we introduce constructors to build elements: $base0$ build the zero function, $\langle 0, 0, \lambda x.0\rangle$, $base2(n, m, i)$ builds the projection, $base2(i)$ builds the successor and $base3$ the predecessor and $base4$ the conditional.

Recursion is allowed "on notation," that is, we can define a function f by recursion as

$$\begin{aligned}
f(0, \overline{x}; \overline{a}) &= g(\overline{x}; \overline{a}) \\
f(iy, \overline{x}; \overline{a}) &= h_i(y, \overline{x}; f(y, \overline{x}; \overline{a}), \overline{a}).
\end{aligned}$$

We introduce the *recursion combinator* $Rec(n, m)$ to accomplish this form of recursion. Its type is

$$(N^n \times N^m \to N) \to (N^{n+1} \times N^{m+1} \to N) \to (N^{n+1} \times N^{m+1} \to N) \to (N^{n+1} \times N^m \to N).$$

The combinator reduces as follows:

$$\begin{array}{ll} Rec(n,m)(g)(h_1)(h_2)(0,\overline{x};\overline{a}) & = g(\overline{x};\overline{a}) \\ Rec(n,m)(g)(h_1)(h_2)(iy,\overline{x};\overline{a}) & = h_i(y,\overline{x};Rec(n,m)(g)(h_1)(h_2)(y,\overline{x};\overline{a}),\overline{a}). \end{array}$$

We also use a "safe composition" combinator $Comp(n,m)$ whose type is

$$(N^n \times N^m \to N) \to (N^n \times N^0 \to N)^n \to (N^n \times N^m \to N)^m \to (N^m \times N^m \to N).$$

Given $h \in N^n \times N^m \to N$, $\bar{r} \in (N^n \times N^0 \to N)^n$ and $t \in (N^n \times N^m \to N)^m$, then

$$Comp(n,m)(h)(\bar{r})(\bar{t})(\bar{x};\bar{a}) = h(\bar{r}(x;\cdot);\bar{t}(\bar{x};\bar{a}))$$

This form of composition can be used to write in combinator form a function expression, say $f(\overline{x};\overline{a})$, in terms of safe composition and projections as long as there is no subexpression $g(\overline{e}_1;\overline{e}_2)$ with a_i appearing among the canonical arguments, $\overline{e}_1$.

Using these constructs we are almost ready to define a class B which will be PTime.

$$\begin{array}{ll} \text{First let } \mathsf{Rec}(B) & = \{f : \mathbb{D} \parallel \exists n,m : N.\ \exists g,h_1,h_2 : B.\ \downarrow(norm(g) = n \\ & \ \&\ safe(g) = m \\ & \ \&\ norm(h_i) = n+1 \\ & \ \&\ safe(h_i) = m+1 \text{ for } i = 1,1 \\ & \ \&\ fun(f) = Rec(n,m)(fun(g))(fun(h_1))(fun(h_2)))\} \\ \mathsf{Comp}(B) & = \{f : \mathbb{D} \parallel \exists n,m : N. \exists h : B. \exists \overline{r} : B^n. \exists \overline{t} : B^m.\ \downarrow(norm(h) = n \\ & \ \&\ safe(h) = m \\ & \ \&\ \forall i : [1,n].\ (norm(r_i) = n \\ & \ \&\ safe(r_i) = 0) \\ & \ \&\ \forall j : [1,m].\ (norm(t_j) = n\ \&\ safe(t_j) = m) \\ & \ \&\ fun(f) = comp(n,m)(fun(h))(fun(\overline{r}))(fun(\overline{t})))\}) \end{array}$$

The functions *fun()* used in these definitions are defined to produce the function part of elements of the recursive type. This is not exactly the same as for the elements of Base since we must account now for the position of Base, Rec(B) and Comp(B) in the disjoint union, but it is a polymorphic function defined independently of the recursive type.

Among the natural constructors to associate with B are the ones for Base, now extended to B, i.e. base0 is modified to wrap "inl" around $\langle 0\langle 0, \lambda p.0\rangle\rangle$ to give $inl(\langle 0, \langle 0, \lambda p.0\rangle\rangle)$. We also want $rec(n,m)(g)(h_1)(h_2)$ and $comp(n,m)(h)(\overline{r})(\overline{t})$ where g, h_1, h_2, h are in B and $\overline{r}, \overline{t}$ are vectors of elements of B.

Def. $B = \mu(B.\ \mathsf{Base} + \mathsf{Rec}(B) + \mathsf{Comp}(B))$

Theorem (Bellantoni & Cook):

- for every $f \in PTime$, there is an $f' \in B$ such that $f(\overline{x}) = f'(\overline{x}; \cdot)$.
- for every $f \in B$, $f(\overline{x}, \overline{y})$ is in PTime.

In proving the Bellantoni and Cook theorem we need to define a complexity measure on elements of B. This is easy because the constructive treatment of inductive types provides with each element of the type, say B, a code showing how it is built. For example, a constructor like

$$\begin{array}{ll} rec(0,1)(base\,1(0,1,2)) & (comp)(0,1)(base\,2(0))(\cdot)(base\,1(0,2,3)) \\ & (comp(0,1)(base\,2(1))(\cdot)(base\,1(0,2,3)) \end{array}$$

not only builds a function in $\mathbb{D}$, but it codes up a complete description of how the element is constructed.

The equality relation on element of B is naturally defined to respect the coding of elements, so it is not extensional function equality. This has always been a feature of inductive definitions, and we now intend to exploit it.

So with each $f \in B$ we can define not only $fun(f)$ but a function $code(f)$ which is the construction history of f. We can think of these codes as an *implicit programming language* that comes with every inductive definition. It is essentially just the expression language of the constructors internalized in some natural way.

other inductive classes

The same technique used to define B can clearly be used to define the elementary functions, E, or the primitive recursive ones, Prim. For any of these classes, the codes provide a way to define resource bounds such as $time(f)$ or $space(f)$.

μ-recursive functions

It is interesting that we can use another natural feature of the Nuprl type theory to define an internal model of the *general recursive functions*. (The interpretation of this result is bound to be provocative.)

Consider the set of functions $\mathbb{F} = \Sigma n : N^+.(N^{n+1} \to N)$ and the subset $\hat{\mathbb{F}}$

$$\{f : \mathbb{F} \mid \exists n : N.\ arity(f) = n + 1 \& \forall x : N^n.\ \exists y : N.f(x, y) = 0\}.$$

On the subset we can define $\mu y.(f(x, y) = 0)$ as the least y such that $f(x, y) = 0$. In Nuprl we can just use *mu* from section 2.2.

Following Kleene [21] we can define the general recursive functions over $\mathbb{F}$, $R(\mathbb{F})$, using this clause

$$\begin{array}{ll} \mathsf{Mu}(C) & = \{f : \mathbb{F} \mid \exists n : N\ \exists g : C.\ arity(f) = n + 1 \\ & \quad \& \downarrow (\forall x : N^n.\ \exists y : N.\ fun(g)(x, y) = 0) \\ & \quad \& fun((f) = \lambda x.\mu y(g(x, y) = 0)\} \\ \\ R(\mathbb{F}) & = \mu(F.\ \mathsf{Base} + \mathsf{Prim\ Rec}(F) + \mathsf{Comp}(F) + \mathsf{Mu}(F)). \end{array}$$

This definition provides an implicit programming language for the general recursive functions. The operation $\downarrow(P)$ is a hiding operation which hides the proof that a y exists.

3.4 Computational complexity measures

Given a set of codes, $Code(\mathcal{C})$, for an inductive definition, we can assign a measure of resource expenditure. In general, this can be any computable function which respects the arity of the specified function, that is arity satisfies

$$fun(f) : A^{(arity(code(f)))} \rightarrow B$$

and a measure satisfies

$$M(code(f)) : A^{arity(code(f))} \rightarrow N.$$

We are only interested in measures which reflect complexity measures that can be imposed on the term structure. (Here we could require that they be Blum measures for example [4].) We want the following faithfulness condition for a measure.

Definition: A measure of computational complexity M on a class $\mathcal{C}(A^* \rightarrow B)$ is faithful iff there is a complexity measure m on term such that for all $f \in \mathcal{C}(A^* \rightarrow B)$, there is a term g such that $fun(f) =_{A^* \rightarrow B} g$ and $M(code(f))(\bar{x}) = m(g)(\bar{x})$.

This condition says that M is actually a measure, a complexity property that is definable on the terms using the evaluation relation of the implementation of the programming language and its logic.

Definition: We define the Time measure on the primitive recursive funtions, over a free algebra A with constructors c_i, say $PR(A^* \rightarrow A)$, in the usual manner [23].

Let
$U_i^n(x_1, \ldots, x_n) = x_i$

$Time(U_i^n) = 1$
$Time(h(g_1(\bar{x}), \ldots, g_n(\bar{x}))) =$
$\sum_{i=1}^{n} Time(g_i)(\bar{x}) + Time(h)(g_1(\bar{x}), \ldots, g_n(\bar{x}))$

$Time(Rg)(c_i(\bar{a}), \bar{x}) =$
$Time(g_{c_i})(Rg(a_1, \bar{x}), \ldots, Rg(a_n, \bar{x}), \bar{a}, \bar{x}) +$
$\sum_{i=1}^{n} Time(Rg)(a_i, \bar{x})$

where Rg is a brief notation for the primitive recursive function definition.

Theorem: Time is a faithful complexity measure on $\Pi\mu\Sigma^+$.

The proof is just a matter of showing that Time mimics the time measure on terms. This can be done because we have picked a class that uses only first order functions. It is more interesting to show that this can be done for inductive classes that use type 2 functions and higher. The result for type 2 can be proved using Melhorn's definitions [25, 7] or those in Bellantoni & Cook [3].

4 Resource Bounded Logics

4.1 Coding logic

As is well known, the predicate calculus can be represented in a type theory such as $\Pi\mu\Sigma^+$ using the propositions-as-types principle. The key points are that the universal quantifier $\forall x : A.\ B$ is represented as a *function space*, $\Pi x : A.\ B$ and the existential quantifier $\exists x : A.\ B$ is represented as $\Sigma x : A.\ B$.

The representation of logic allows us to state program specifications as propositions to be constructively proved. For example, the problem of finding the least prime factor of a number can be stated using $\{2 \cdot\cdot\} = \{x : N \mid 2 \leq x\}$:

$$\forall n : \{2 \cdot\cdot\}\ .\ \exists p : N.\ \ p \text{ is the least prime factor of } n$$

We would like to be able to put a *complexity constraint* on this specification, asking that p be found in polynomial time say.

4.2 Resource bounded logics

We want to define the class of feasible proofs in type theory. The major criterion is that for the types needed in number theory (specifically HA^ω), say A and B, if we feasibly prove $\forall x : A.\ \exists y : B.\ R(x, y)$, then $\exists f : Poly(A \rightarrow B).\forall x : A.R(x, f(x))$ where $Poly(A \rightarrow B)$ are the polynomial time functions from A to B. The definition of $Poly(A \rightarrow B)$ follows the approach to the complexity of higher-order operators taken in Constable[7], Mehlhorn [25], Cook & Urquant [11].

In order to define feasible proofs, we use the propositions-as-types principle and define $\forall x : A.\exists y B.R(x, y)$ as the function type $\Pi(A; x.\Sigma(B.y.R(x, y)))$ (or in Nuprl notation, $x : A \rightarrow y : B \times R(x, y)$). We need to define a notion of polynomial time function in the dependent function class $\Pi(A; x.B)$. We follow the method of defining complexity bounds over elements of an inductively defined class large enough to include the objects we want.

numerical types

We first define the types needed for number theory (HA^ω).

$$\begin{aligned}\mathcal{T} = \mu(T.\quad & \{A : U_1 \,||\, A = \mathbb{N} \vee A = void \vee \exists\, n, m : \mathbb{N}. A = Eq(n,m)\} + \\ & \{A : U_1 \,||\, \exists\, B_1, B_2 : T. \downarrow (A = ty(B_1) \times ty(B_2))\} + \\ & \{A : U_1 \,||\, \exists\, B_1, B_2 : T. \downarrow (A = ty(B_1) + ty(B_2))\} \\ & \{A : U_1 \,||\, \exists\, B_1, B_2 : T. \downarrow (A = ty(B_1) \rightarrow ty(B_2))\} + \\ & \{A : U_1 \,||\, \exists\, B : T. \exists P : ty(B) \rightarrow U_1. \downarrow (A = \Sigma x : ty(B).\ P(x))\} \\ & \{A : U_1 \,||\, \exists\, B : T. \exists P : ty(B) \rightarrow U_1. \downarrow (A = \Pi x : ty(B). P(x))\})\end{aligned}$$

Notice that

$$\{A : U_1 \mid \exists T : \mathcal{T}.\ A = ty(T)\} \subseteq U_1.$$

Let $\mathcal{F} = \Sigma T : \mathcal{T}.\ T$; the elements are pairs $\langle T, t\rangle$ where $T \in \mathcal{T}$ and $t \in T$. Our goal is to define the subclass of "polynomial time" elements of $\mathcal{F}$, called $Poly(\mathcal{F})$. We will define this as a subtype of inductively defined class called $\mathcal{R}(\mathcal{F})$, using the measure of computational complexity for higher type objects mentioned above [7, 25, 11].

We will define $\mathcal{R}(\mathcal{F})$ to be the least class containing elements of $\mathbb{N}$ and "elements" (proofs) of the atomic types, the successor functions, the identity functions, and closed under pairing, selection, composition, constants, explicit operations on arguments (permutation of arguments, duplication of arguments and application) and primitive recursion. By the results of Grzegorczyk [17] this class will include all the primitive recursive objects of higher type. Then we define complexity functions and restrict $\mathcal{R}(\mathcal{F})$ to the polynomial time computable objects of finite type, $Poly(\mathcal{F})$. The details follow the pattern of 3.3 and are omitted here. Basically $\mathcal{R}(\mathcal{F})$ is

$$\mu(F.\ Base + Constants(F) + Pairs(F) + Explicit(F) + Recursion(F)).$$

Once we can define $Poly(\mathcal{T})$ it is possible to express the condition we needed above using propositions-as-types. For any type $T \in \mathcal{T}$, let $Poly(T)$ be those functions of $Poly(\mathcal{T})$ of type T. Then

$$\forall_{Poly} x : N.\ P(x) == Poly(\Pi x : N.\ P(x)).$$

In general for a complexity class $\mathcal{C}^T(\Pi x : A.\ P(x))$ we can define

$$\forall_{CT} x : A.\ P(x) = \mathcal{C}^T(\Pi x : A.\ P(x)).$$

5 Acknowledgments

I would like to thank my colleagues Stuart Allen and Chetan Murthy for discussions about the ideas of this paper and Kate Ricks for preparing the manuscript.

References

[1] S. F. Allen, R. L. Constable, D. J. Howe, and W. Aitken. The Semantics of Reflected Proof. In *Proc. of Fifth Symp. on Logic in Comp. Sci.*, pages 95–197. IEEE, June 1990.

[2] J. L. Bates and R. L. Constable. Proofs as programs. *ACM Trans. Program. Lang. and Syst.*, 7(1):53–71, 1985.

[3] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[4] M. Blum. A machine independent theory of computational complexity. *J. ACM*, 14:322–336, 1967.

[5] S. Buss. The polynomial hierarchy and intuitionistic bounded arithmetic. In *Structure in Complexity Theory, Lecture Notes in Computer Science No. 223*, pages 77–103. Springer-Verlag, 1986.

[6] A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proc. of the 1964 Int. Congress for Logic, Methodology, and Phil. of Sci.*, pages 24–30, North-Holland, 1965.

[7] R. L. Constable. Type two computational complexity. In *Proc. 5th ACM Symp., Theory of Computing*, pages 108–121, 1973.

[8] R. L. Constable, S. F. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. J. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[9] R. L. Constable and N. Mendler. Recursive Definitions in Type Theory. In *Proc. of Logics of Prog. Conf.*, pages 61–78, January 1985. (Cornell TR 85-659).

[10] R. L. Constable and D. Zlatin. The type theory of PL/CV3". *ACM Trans. on Prog. Lang. and Systems*, 6(1):94–117, January, 1984.

[11] S. Cook and A. Urquhart. Functional interpretations of feasibly constructive arithmetic. In *Proceedings of 21st Symposium on the Theory of Computing*, pages 107–112. ACM, 1989. To appear in *Annals of Pure and Applied Logic*.

[12] S. A. Cook and B. M. Kapron. Characterizations of the basic feasible functionals of finite type. In S. Buss and P. Scott, editors, *Proceedings of MSI Workshop on Feasible Mathematics*, pages 71–95, New York, 1990. Birkhauser-Boston.

[13] N. G. deBruijn. The mathematical language Automath, its usage and some of its extensions. In *Symp. on Automatic Demonstration,* Lecture Notes in Mathematics, Vol. 125, pages 29–61. Springer-Verlag, 1970.

[14] P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

[15] S. Feferman. A language and axioms for explicit mathematics. In *Algebra and Logic,* Lecture Notes in Mathematics, pages 87–139. Springer-Verlag, 1975.

[16] J.-Y. Girard, A. Scedrov, and P. J. Scott. Bounded linear logic. *Theoretical Computer Science*, 97(1):1–66, 1992.

[17] A. Grzegorczyk. Recursive objects in all finite types. *Fundamenta Mathematicae*, 54:73–93, 1964.

[18] J. Hartmanis and R. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematics Society*, 117:285–306, 1965.

[19] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, NY, 1980.

[20] D. J. Howe. On computational open-endedness in Martin-Löf's type theory. In *Proc. of Sixth Symp. on Logic in Comp. Sci.*, pages 162–172. IEEE Computer Society, 1991.

[21] S. C. Kleene. *Introduction to Metamathematics.* D. Van Nostrand, Princeton, 1952.

[22] D. Leivant. Functions over free algebras definable in the simply typed lambda calculus. *Theoretical Computer Science*, 121:309–321, 1993.

[23] D. Leivant. Stratified functional programs and computational complexity. In *Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 325–333, Charleston, SC, January 1993. ACM, ACM Press.

[24] P. Martin-Löf. *Intuitionistic Type Theory, Studies in Proof Theory, Lecture Notes.* Bibliopolis, Napoli, 1984.

[25] K. Mehlhorn. Polynomial and abstract subrecursive classes. *Journal of Computer and System Sciences*, pages 148–176, 1976.

[26] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the calculus of constructions. In *Mathematical Foundations of Program Semantics, 5th International Conference,* Lecture Notes in Computer Science, Vol. 442, pages 209–228. Springer-Verlag, 1989.

[27] D. Scott. Data types as lattices. *SIAM J. Comput.*, 5:522–87, 1976.

[28] S. Smith and R. L. Constable. Partial objects in constructive type theory. In *Proc. of Second Symp. on Logic in Comp. Sci.*, pages 183–93. IEEE, Washington, D.C., 1987.

[29] K. Weihrauch. A simple and powerful approach for studying constructivity, computability and complexity. In J. Myers and M. O'Donnell, editors, *Constructivity in Computer Science, Logic in Computer Science 613*, pages 228–246. Springer-Verlag, 1991.