Abstract Complexity in Type Theory

Ryan Williams *

1 Introduction

We develop an abstract complexity theory based on type-theoretical programming language semantics. We provide two axioms for complexity measures in type theory that have direct analogues with Blum's axioms [2] for classifying the complexity of partial recursive functions. However, our treatment is strictly more general than Blum's in several respects. Rather than being defined in terms of a family of partial recursive functions on the natural numbers, our complexity measures are functions from reflected terms to actual terms in the language. We show our definition captures Blum's in several striking ways; we demonstrate proofs of recursive relatedness, arbitrarily complex programs for a function, infinite complexity hierarchies, and a gap theorem [3] within our formulation. Furthermore, following previous work of Constable and Crary [6, 7], our formulation of abstract complexity elucidates the distinction between expressions as functions and expressions as algorithms, i.e. extensional semantics versus intensional.

Previous work bringing together type theory notions and complexity has focused on finding model-independent characterizations of certain types of resource bounds, such as the polynomial time or polynomial space computable functions (for example, [13, 1]; cf. [9] for a survey of programming languages that precisely express various complexity concepts). We have not encountered in the literature an abstract treatment of complexity in a type theoretic environment, though there are a scant number of relatives. As a reference, Jones [10] gives a nice overview of complexity using tools from programming language semantics, proving several of Blum's theorems in the special case where runtime is the complexity measure. Constable and Borodin [4] formulated many of the original abstract complexity theorems within a simple programming language.

2 Notation/Type Theory

The type theory we will use is a partial object type theory introduced by Constable and Crary [6], which has also been implemented in the Nuprl system. The theory has two novel aspects: *partial types*, and a small *reflection* mechanism. We will not discuss it in full here; instead we will go over some basic ideas of it that will be used in this paper.

$\mathbb{N}$ is the type for natural numbers, $\mathbb{B}$ is the type Boolean. From ground types T_1 and T_2 such as these we build extended types such as product types $T_1 \times T_2$, function types $T_1 \to T_2$, and subset types $\{t : T \mid P(t)\}$, where $P : T \to \mathbb{B}$.

The basic theory has a computation system that is summarized in [6]. (In the final conference submission, we will include the full system or at least a substantial subset; for now we will not repeat

*Dept. of Computer Science, Cornell University, Ithaca, NY 14850. Email: `rrw9@cornell.edu`. Supported by an NSF Graduate Research Fellowship.

it in its entirety.) For simplicity of presentation, our system will include an if...then...else construct, which can be easily defined in the operational semantics. An expression in the computation system is called a term, and we use the variable t to denote one. The operational semantics define a small step relation $t_1 \mapsto t_2$ from one term to the next. We say $canon(t)$ if for $t' \in term$, $[t' \neq t \Rightarrow \neg(t \mapsto t')]$; i.e. t is a "value" and not an "expression" to be evaluated. As usual, the transitive closure of $\mapsto$ is $\mapsto^*$, and we say $t \downarrow$ (t converges) if there is a t' such that $t \mapsto^* t'$ and $canon(t')$; otherwise, we say $t \uparrow$ (t diverges).

Every term of the language is of type *term*. Define

$$value = \{t : term \mid canon(t)\},$$

so it is the subset type of *term* with all canonical terms. So for example, $((\lambda x : \mathbb{N}.\ x)\ 3)$ is not of type *value*, but $(\lambda x : \mathbb{N}.\ x)$ is.

2.1 Partial Types

Partial types were introduced by Constable and Smith [5] (with related work by Moggi [12]) to reason about partial recursive functions in a type theoretic setting; that is, functions that do not necessary halt on all inputs. Partial types may be thought of as types that include all non-halting computations, or rather, objects that never fully evaluate to a value. If T is a type, $\overline{T}$ denotes the corresponding partial type, which (roughly speaking) includes objects of T treated as convergent computations (if $t : T$ then $t \downarrow$, $t : \overline{T}$), and a divergent computation (there is $t_\infty : \overline{T}$ with $t_\infty \uparrow$). Equality is defined as

$$t_1 = t_2 : \overline{T} \iff (t_1 \uparrow\ \wedge\ t_2 \uparrow)\ \vee\ (t_1 \downarrow\ \wedge\ t_2 \downarrow\ \wedge\ t_1 = t_2 : T).$$

If T is not a partial type (all of its objects are convergent) then we call T a *convergent type*.

In conjunction with partial types, a fixed point operator is used in the language. For any $f : \overline{A} \to \overline{A}$, $fix(f) : \overline{A}$, and $f(fix(f)) \mapsto fix(f)$. To be valid, the fixed point operator inference requires that the type A be *admissible*. Crary [8] has shown that admissibility is preserved for a very wide range of types, which includes the vast majority of types considered in practice. It is also the case that quoted types are also admissible if what's being quoted is admissible.

2.2 Reflection

The computation system also has a limited type of reflection, or a "quoting" mechanism. All of the *terms* have corresponding reflected Terms. For each $t : term$ there is a unique corresponding object $\ulcorner t \urcorner : \mathsf{Term}$. The object $\ulcorner t \urcorner$ is called the *quote of* t. We will denote elements of Term by either quoted objects of *term* or in block font with no quotes, i.e. $\mathsf{a}, \mathsf{b}, \ldots, \mathsf{t}, \ldots$.

It is instructive to think of the object $\ulcorner t \urcorner$ in Term as an algorithm implementing precisely the term meant by $t : term$. Note that two terms t_1 and t_2 may evaluate to the same term t_3, and thus $t_1 = t_2 \in term$, however these terms may be different syntactically; i.e. $\ulcorner t_1 \urcorner \neq \ulcorner t_2 \urcorner \in \mathsf{Term}$.

This system of reflection also nudges us to introduce another kind of extended type. When T is already a type, we define the quoted type

$$[T] := \{\ulcorner t \urcorner \mid t : T\}.$$

Thus $[T]$ is simply the type of quoted objects of T.

3 Blum's notion of measure

In this section we briefly review Blum's axioms in abstract complexity. Blum [2] formulated two axioms for what a complexity measure on partial recursive functions should be like. Given a list $\{\phi_i\}$ of partial recursive functions on $\mathbb{N}$, we consider an ensemble of partial recursive functions $\{B_i\}$ on $\mathbb{N}$ corresponding to the ϕ_i. The ensemble satisfies the axioms:

1. $\forall n : \mathbb{N}.\ \ \phi_i(n) \downarrow \iff B_i(n) \downarrow$, and
2. $L = \{\langle i, n, m \rangle : B_i(n) = m\}$ is recursive (i.e. decidable).

Blum defines characteristic functions M for L to be *measures*. To avoid confusion, we will call them *Blum measures*.

Our goal has been to extend this concept of measure to a type theoretic setting, being as general as possible while still being able to prove results from Blum's original paper. Next we will begin to demonstrate our progress towards this goal.

4 Abstract measures on types

We start by giving an extremely abstract notion of typed measure, then taking a special case of the definition to deal with Blum's idea of a measure. This abstract definition may be interesting to study in its full generality, but we will not do that here.

Definition 4.1 *A function m from* Term *to* $\overline{value}$ *is a typed measure if*

$$\forall \ulcorner t_1 \urcorner : \mathsf{Term}.\ \ t_1 \downarrow \iff (m \ulcorner t_1 \urcorner) : value \tag{1}$$

$$\exists g : \mathsf{Term} \times value \to \mathbb{B}.\ \forall \ulcorner t_1 \urcorner : \mathsf{Term}.\ \forall t_2 : value.\ \ (g \ulcorner t_1 \urcorner\ t_2) = 1 \iff (m \ulcorner t_1 \urcorner) = t_2. \tag{2}$$

So a typed measure sends reflected terms to actual canonical terms; every expression of the language is measurable. The first axiom says that m is defined just in case the term t_1 converges to a canonical term. The second says that it can be always determined if $m(\ulcorner t_1 \urcorner) = t_2$ for a particular t_1 and t_2 (note $\mathbb{B}$ is not a partial type, so g is posited as total recursive).

Typed measures can return other measures. For the purposes of our paper, we simply wish to model the old familiar concepts of abstract measure in a type theoretic setting. So we will restrict the above definition to explicitly capture Blum's notion of a complexity measure as involving a family of partial recursive functions.

Before we do this, we will need to define ordered types, in order to strike the aforementioned desired balance between generality and provability. The theorems of abstract complexity on natural numbers liberally use the fact that $\mathbb{N}$ is a totally ordered set with a least element; we will also need such a property for proving abstract complexity theorems within types. It is interesting to see what restrictions from the old results can be lifted without affecting correctness.

4.1 Totally ordered types

Blum's theorems crucially use the fact that the co-domains of the B_i functions are of the same type and are totally ordered, so that we may say there is (for example) no "fastest" program for some function under a measure, or there are always functions that are "more complex" than others. For this reason, we give the following definition. If T is a type, define the extended type $T^{t_0}_{\leq}$ to be the type T with a total order $\leq$ on all of its objects, and a least element $t_0 : T$.

If T is a partial type, then along with being reflexive, anti-symmetric, and transitive, we require that the total order satisfies one extra property:

$$(t, t' \in T) \wedge (t \downarrow) \wedge (t' \uparrow) \Rightarrow (t \leq t').$$

That is, divergent computations are greater than convergent ones. Since all divergent computations are treated as equal in T (i.e. $t \uparrow$ and $t' \uparrow$ implies $t = t' \in T$), this means T has a greatest object in the ordering as well; namely, a divergent term.

We'll call $T^{t_0}_{\leq}$ a *totally ordered type*. When several totally ordered types are involved at once, we will use $\leq_T$ to refer to the ordering on objects of $T^{t_0}_{\leq}$.

As we will see later, we need Term (the quotes of terms in the language) to be totally ordered with a least element. There are several canonical ways to order objects of Term, and any of them are suitable for our purposes, so we refrain from giving a particular ordering here. Whatever ordering is chosen, in the sequel we will denote it by $\leq_q$.

4.2 Typed Blum measures

We first give the definition of a typed Blum measure, then explain its origin. For types T_1 and T_2, define

$$[T_1, T_2] := \{\ulcorner(f\ x)\urcorner \mid f : T_1 \to T_2,\ x : T_1\}.$$

In other words, $[T_1, T_2]$ is Term restricted to quotations of expressions about to be evaluated, for the specific types T_1 and T_2.

Definition 4.2 *Let A, B be types and let C be a totally ordered type. A function $m : [A \to B] \to ([A] \to \overline{C})$ is a* typed Blum measure *if $m' : [A, B] \to \overline{C}$ is a measure, and for all $\ulcorner f \urcorner : [A \to B]$ and $\ulcorner a \urcorner : [A]$, $m(\ulcorner f \urcorner)(\ulcorner a \urcorner) = m'(\ulcorner(f\ a)\urcorner)$.*

The purpose of defining m in terms of another measure m' is that our original definition of measure takes arbitrary quoted expressions as input, but Blum's measure may be thought of as having two inputs: a function label and a label for the input.

We will see that this definition is sufficient for proving several of Blum's theorems such as recursive relatedness and the existence of infinitely many complexity classes. We believe the gap and speed-up theorems can also be proven within this framework, as their proofs differ only in their diagonalization strategies. All we require is that m maps quoted functions to functions that take quoted inputs and produce objects from a type with an ordering.

The correspondence to Blum's original definition of measure may be seen through a simple analogy. Consider $\ulcorner f \urcorner : [\mathbb{N} \to \mathbb{N}]$, $\ulcorner x \urcorner : [\mathbb{N}]$, and a typed Blum measure $m : [\mathbb{N} \to \mathbb{N}] \to ([\mathbb{N}] \to \overline{C})$ by the definition given above. Then for any Blum measure $M(i, n, m)$ in partial recursive function theory there is a corresponding typed Blum measure m in our theory such that

$$m(\ulcorner f \urcorner)(\ulcorner x \urcorner) = t \iff M(i, n_x, n_t) = 1,$$

where $\Phi_i = f$, n_x is the Gödel number of x, and n_t is the Gödel number of t.

5 Time and Space Complexity

Here we show how our formulation of measure fits in with previous work. Constable and Crary define predicates $Time(e,t)$ and $Space(e,s)$ for determining the time and space complexity of arbitrary terms. We give the predicates here without defining all of the notation involved with them, assuming it is straightforward to deduce.

$$\begin{aligned} Time(e,t) \iff \exists n:[0\cdots t].\ \exists f:[0\cdots n]\to \mathsf{Term}. \quad & [f(0)=e \ in\ \mathsf{Term}\ \wedge \\ & iscanon(f(n)) = true\ in\ \mathbb{B} \wedge \\ & \forall i:[0\cdots(n-1)].\ f(i)\ evalto\ f(i+1)]. \end{aligned}$$

This predicate is true iff the term e becomes a canonical term after t steps of evaluation. Note that since evaluation is deterministic, for every $\ulcorner e \urcorner : \mathsf{Term}$ there is at most one t such that $Time(\ulcorner e \urcorner, t)$ is true. This leads us to define a function $time : \mathsf{Term} \to \overline{N}$ with the property that

$$time(\ulcorner e \urcorner) = t \iff \mathsf{Term}(\ulcorner e \urcorner, t).$$

Assuming $time$ can be defined within our language (and it can), it is easy to check that $time$ is a typed measure in the abstract sense: $e\downarrow \iff \exists t:\mathbb{N}.\ Time(\ulcorner e \urcorner, t) \iff time(\ulcorner e \urcorner):\mathbb{N} \iff time(\ulcorner e \urcorner)\downarrow$, so Axiom 1 is satisfied. Also, $time(\ulcorner e \urcorner) = t \iff Time(\ulcorner e \urcorner, t)$, but $Time$ is a recursive predicate, hence Axiom 2 is satisfied.

Next, let $size : \mathsf{Term} \to \mathbb{N}$ be a function mapping the quoted term t to the number of symbols in t. Then a space complexity predicate may be defined as

$$\begin{aligned} Space(e,s) \iff \exists n:\mathbb{N}.\ \exists f:[0\cdots n]\to \mathsf{Term}. \quad & [f(0)=e \ in\ \mathsf{Term}\ \wedge \\ & iscanon(f(n)) = true\ in\ \mathbb{B} \wedge \\ & \forall i:[0\cdots(n-1)].\ f(i)\ evalto\ f(i+1) \wedge \\ & \forall i:[0\cdots n].\ size(f(i)) \leq s]. \end{aligned}$$

This predicate actually cannot be captured via our definition of a typed measure, as it stands. In Blum's theory, he defines the space usage of a divergent computation to be infinite; otherwise, the space usage of a term may be defined while the term is divergent, violating Axiom 1. Defining space usage in a similar way here would allow us to satisfy the axioms.

6 Abstract Complexity Results

Here we will present theorems proving the major abstract complexity results in our typed complexity setting. In some cases, the old proofs given for partial recursive functions immediately translate over to type theory. For cases where diagonalization is necessary, they do not; however, we end up with simpler proofs of the same results. The first theorem shows that any two typed measures are recursively related almost everywhere. Its proof is of the first kind– a translation from the original.

Theorem 6.1 (Blum, Theorem 3) *Let C be a convergent type, and let $m_1, m_2 : [A \to B] \to ([A] \to \overline{C})$ be Blum measures. Then there exists $r : C \to ([A] \to C)$ such that for all $f : A \to B$, there is an $\mathsf{a} \in [A]$ with*

$$(r\ m_1(\ulcorner f \urcorner)(\mathsf{a}'))(\mathsf{a}') \geq_C m_2(\ulcorner f \urcorner)(\mathsf{a}')$$

for all $\mathsf{a}' \geq_A \mathsf{a}$.

Proof. Note the absence of a bar over C indicates that r is claimed to be recursive. Define

$$r = \lambda c : C.\ \lambda \mathsf{a} : [A].\ \max_{\ulcorner f \urcorner \leq_q \mathsf{a}} \{m_2(\ulcorner f \urcorner)(\mathsf{a})\ :\ m_1(\ulcorner f \urcorner)(\mathsf{a}) = c\}.$$

A maximum can easily be seen to be constructed in our language, but it is tedious so we omit the construction. We claim r is recursive. First, the predicate $[m_1(\ulcorner f \urcorner)(\mathsf{a}) = c]$ can be recursively decided (Axiom 2). Secondly, if it is true, then $(f\ a) \downarrow$ by Axiom 1, therefore $m_2(\ulcorner (f\ rep(\mathsf{a})) \urcorner) \in C$, again by Axiom 1. Also, for any $f : A \to B$, let $\mathsf{a} : [A]$ be such that $\mathsf{a} >_q \ulcorner f \urcorner$; then for any $\mathsf{a}' \geq_q \mathsf{a}$,

$$\begin{aligned}(r\ m_1(\ulcorner f \urcorner)(\mathsf{a}'))(\mathsf{a}') &= \max_{\ulcorner f \urcorner \leq_q \mathsf{a}'} \{m_2(\ulcorner f \urcorner)(\mathsf{a}')\ :\ m_1(\ulcorner f \urcorner)(\mathsf{a}') = m_1(\ulcorner f \urcorner)(\mathsf{a}')\} \\ &= \max_{\ulcorner f \urcorner \leq_q \mathsf{a}'} m_2(\ulcorner f \urcorner)(\mathsf{a}') \geq m_2(\ulcorner f \urcorner)(\mathsf{a}'),\end{aligned}$$

since the maximum is being taken. □

Looking over the proof, one can see why C and Term need to be totally ordered. The next result is also in the spirit of the original one given by Blum. It shows that a poorly described function can "waste a huge number of steps in computing a simple function" [2]. Before that we give a definition needed to perform the diagonalization in the proof.

Definition 6.1 *A* contrary map *is a $q : B \to B$ such that for all $b : B$, $q(b) \neq b$.*

It is easy to see that we can construct such a q for any reasonable type with two or more elements. In the following we assume A and B are admissible, for the purposes of using the fix operator.

Theorem 6.2 (Blum, Theorem 4) *Let A be an admissible type, and B a totally ordered admissible type with a contrary map q. For any $h : [A] \to B$, $f : A \to B$, and typed Blum measure $m : [A \to B] \to ([A] \to B)$, there is a $g : A \to B$ such that $g = f : A \to B$ and for all $\mathsf{a} : [A]$,*

$$m(\ulcorner g \urcorner)(\mathsf{a}) >_B h(\mathsf{a}).$$

Proof. Define

$$\hat{\mathsf{g}} = \lambda \mathsf{g} : [A \to B].\ulcorner \lambda a : A.\ \mathsf{if}\ [m(\mathsf{g})(\ulcorner a \urcorner) >_B h(\ulcorner a \urcorner)]\ \mathsf{then}\ f(a)\ \mathsf{else}\ q(ref(\mathsf{g})(a))\urcorner.$$

Note that $\hat{\mathsf{g}}$ is total recursive, so $\hat{\mathsf{g}} : [A \to B] \to [A \to B]$.

Consider $fix(\hat{\mathsf{g}}) : [A \to B]$. ($[A \to B]$ is admissible.) If $m(fix(\hat{\mathsf{g}}))(\ulcorner a \urcorner) \leq h(\ulcorner a \urcorner)$ for some $a : A$, then

$$ref(fix(\hat{\mathsf{g}}))(a) = q(ref(fix(\hat{\mathsf{g}}))(a)),$$

contradicting the properties of q. Therefore for all $\mathsf{a} : [A]$, not only is

$$m(fix(\hat{\mathsf{g}})(\mathsf{a}) >_B h(\mathsf{a}),$$

but also for all $a : A$,

$$ref(fix(\hat{\mathsf{g}}))(a) = f(a).$$

Thus $ref(fix(\hat{\mathsf{g}}))$ is the desired function. □

6.1 Complexity Hierarchies and a General Diagonalization Condition

The next results we show have proofs that are cleaner and more general than those normally given in abstract complexity. The first of these results is one demonstrating an infinite hierarchy of complexity classes, for any typed Blum measure m. To this end we will give a general definition of conditions for a subset type to be "diagonalizable". Given these conditions, one can always construct a function that is not in the subset type but in the original type.

Definition 6.2 *Let $T_P = \{f : [A \to B] \mid P(f)\}$ be a subset type of $[A \to B]$. A total recursive and onto $r : [A] \to T_P$ is a* cross-labeler.

Definition 6.3 *Let $T_P = \{f : [A \to B] \mid P(f)\}$ be a subset type of $[A \to B]$. T_P is* diagonalizable over $[A \to B]$ *if T_P has a cross-labeler r and B has a contrary map q.*

From this definition, we prove a simple lemma that shows if a subset type is diagonalizable, then we can construct a function that is "hard" for the subset type. As the definition suggests, the argument is a traditional diagonalization.

Lemma 6.1 *Suppose $T_P = \{f : [A \to B] \mid P(f)\}$ is diagonalizable over $[A \to B]$. Then there exists $\ulcorner h \urcorner : [A \to B]$ that is not in T_P.*

Proof. Let r and q be the corresponding functions existing due to diagonalizability of T_P. Consider the function $h = \lambda a : A.\ q((ref(r(\ulcorner a \urcorner)))(a))$. Observe that $h : A \to B$. Now for sake of contradiction, suppose $\ulcorner h \urcorner \in T_P$. Then because r is onto, there is an $a_h : A$ with $r(\ulcorner a_h \urcorner) = \ulcorner h \urcorner$. This implies $h(a_h) = q((ref(r(\ulcorner a_h \urcorner)))(a_h)) = q(ref(\ulcorner h \urcorner)(a_h)) = q(h(a_h))$, which contradicts the definition of q. □

We now formulate "infinitely-often" complexity classes, in order to show there are arbitrarily complex functions for any typed Blum measure. Let A be a totally ordered type. Let $m : [A \to B] \to ([A] \to \overline{C})$ be a typed Blum measure and $g : [A] \to C$. Define

$$[A \to B]_{m \leq g \text{ i.o.}} := \{\mathsf{f} : [A \to B] \mid \forall \mathsf{a} : [A].\ \exists \mathsf{a}' : [A].\ [\mathsf{a} \leq_q \mathsf{a}' \land m(\mathsf{f})(\mathsf{a}') \leq_C g(\mathsf{a}')]\}.$$

This represents the class of functions under complexity m that is bound from above infinitely often by g. An analogous type may be defined for bounds that are almost everywhere:

$$[A \to B]_{m \leq g \text{ a.e.}} := \{\mathsf{f} : [A \to B] \mid \exists \mathsf{a} : [A].\ \forall \mathsf{a}' : [A].\ [\mathsf{a} \leq_q \mathsf{a}' \Rightarrow m(\mathsf{f})(\mathsf{a}') \leq_C g(\mathsf{a}')]\}.$$

The following result is not hard to show, and using the above lemma it implies that for functions over the natural numbers, there is an infinite hierarchy of types with more complex functions.

Theorem 6.3 *For any typed Blum measure m and (total recursive) g, $[\mathbb{N} \to \mathbb{N}]_{m \leq g \text{ i.o.}}$ is diagonalizable over $[\mathbb{N} \to \mathbb{N}]$.*

Corollary 6.1 (Blum, Theorem 7) *For $g : [\mathbb{N}] \to \mathbb{N}$, there exists a function $h : \mathbb{N} \to \mathbb{N}$ such that*

$$m(\ulcorner h \urcorner)(\ulcorner a \urcorner) > g(\ulcorner a \urcorner)$$

almost everywhere.

6.2 Gap Theorem

The final theorem we present is an analogue of the famous gap theorem, independently discovered by both Borodin [3] and Blum [2]. Let $m : [A \to B] \to ([A] \to \overline{C})$ be a Blum measure. (So C is totally ordered.) Our proof is not too different from a traditional proof of the gap theorem, though certainly more general.

Theorem 6.4 (GAP THEOREM) *Let $g : C \to C$ with $g(c) \geq_C c$ for all $c : C$. If $[A \to B]_{m \leq (g \circ f)\ a.e.}$ has a cross-labeler, then there is $f : [A] \to C$ such that*

$$h : [A \to B]_{m \leq (g \circ f)\ a.e.} \Rightarrow h : [A \to B]_{m \leq f\ a.e.}.$$

Proof. Let $r : [A] \to [A \to B]_{m \leq (g \circ f)\ a.e.}$ be the cross-labeler. First, we define a predicate gap on $[A] \times C$.

$$\mathsf{gap}(\mathsf{a}, c) \iff \forall \mathsf{a}' \leq_q \mathsf{a} : [A].[m(r(\mathsf{a}'))(\mathsf{a}) >_C g(c) \ \vee \ m(r(\mathsf{a}'))(\mathsf{a}) \leq_C c)].$$

Now define

$$f = \lambda \mathsf{a} : [A].\ \min_{c:C}\{\mathsf{gap}(\mathsf{a}, c)\}.$$

As with max in Theorem 6.1, min can be defined within the theory, but we omit its construction here. The same holds for the gap predicate, which is recursive.

We claim f is total recursive; let $c' = \max_{\mathsf{a}' \leq_q \mathsf{a}:[A]} m(r(\mathsf{a}'))(\mathsf{a})$. This maximum is defined (finite) since we are only considering functions $r(\mathsf{a}') : [A \to B]_{m \leq (g \circ f)\ a.e.}$, i.e. of bounded measure. Furthermore $c' : C$ is an upper bound on the minimum $c : C$ satisfying $\mathsf{gap}(\mathsf{a}, c)$ because $m(r(\mathsf{a}'))(\mathsf{a}) \leq_C c'$ for all $\mathsf{a}' \leq_q \mathsf{a}$, by definition.

Now consider any $\ulcorner h \urcorner : [A \to B]_{m \leq (g \circ f)\ a.e.}$, so $\exists \mathsf{a}^{\mathsf{h}} : [A].\ \forall \mathsf{a}' \geq_q \mathsf{a}^{\mathsf{h}}.\ m(\ulcorner h \urcorner)(\mathsf{a}') \leq g(f(\mathsf{a}'))$. Let a_{h} be such that $r(\mathsf{a}_{\mathsf{h}}) = \ulcorner h \urcorner$. Set $\mathsf{a}^* = \max\{\mathsf{a}_{\mathsf{h}}, \mathsf{a}^{\mathsf{h}}\}$. Then for all $\mathsf{a}' \geq_q \mathsf{a}^*$, $f(\mathsf{a}')$ satisfies

$$[m(r(\mathsf{a}_{\mathsf{h}})(\mathsf{a}') = m(\ulcorner h \urcorner)(\mathsf{a}') >_C g(f(\mathsf{a}'))] \ \vee \ [m(\ulcorner h \urcorner)(\mathsf{a}') \leq_C f(\mathsf{a}')],$$

but since the first disjunction is never true ($\mathsf{a}' \geq_q \mathsf{a}^{\mathsf{h}}$), we have that $m(\ulcorner h \urcorner)(\mathsf{a}') \leq_C f(\mathsf{a}')$ for all $\mathsf{a}' \geq_q \mathsf{a}^*$. That is, $\ulcorner h \urcorner : [A \to B]_{m \leq f\ a.e.}$. □

7 Applications

We have presented a relatively clean approach to abstract complexity theory. It is appealing in several ways: it ties together ideas from programming language semantics and complexity intricately, it is implementable (via Nuprl), and its proofs are quite general and elegant in their own right.

We believe that by using similar methods to those presented, one can prove a tight time and space hierarchy of functions within the type theory. This may be doable by measuring the overhead provided by the "diagonalizing function" $h = \lambda a : A.\ q((ref(r(\ulcorner a \urcorner)))(a))$ from Theorem 6.1. In addition, ideas from Kozen's axiomatic work on the power of diagonalization [11] may be translated over to this setting, which could admit further progress in the development of complexity in our language.

This work is still in progress. In our final conference submission, we will include more detail concerning the type theory itself.

References

[1] S. Bellantoni, K.-H. Niggl, and H. Schwichtenberg. *Higher Type Recursion, Ramification and Polynomial Time*, Annals of Pure and Applied Logic 104, 2000.

[2] M. Blum. *A machine-independent theory of the complexity of recursive functions.* Journal of the ACM 14(2):322-336, 1967.

[3] A. Borodin. *Computational complexity and the existence of complexity gaps.* Journal of the ACM 19(1):158-174, 1972.

[4] R. L. Constable and A. Borodin. *Subrecursive Programming Languages, Part I: efficiency and program structure.* Journal of the ACM 19(3): 526-568, 1972.

[5] R. L. Constable and S. F. Smith. *Computational foundations of basic recursive function theory.* Theoretical Computer Science 121: 89-112, 1993.

[6] R. L. Constable and K. Crary. *Computational complexity and induction for partial computable functions in type theory.* Technical report, Department of Computer Science, Cornell University, 1997.

[7] R. L. Constable. *A note on complexity measures for inductive classes in constructive type theory.* Information and Computation 143(2):137-153, 1998.

[8] K. Crary. *Admissibility of fixpoint induction over partial types.* Conference on Automated Deduction, 270-285, 1998.

[9] M. Hofmann. *Programming languages capturing complexity classes.* SIGACT News Logic Column 9, 2000.

[10] N. D. Jones. *Computability and Complexity: From a Programming Perspective.* MIT Press, 1997.

[11] D. Kozen. *Indexing of subrecursive classes.* Proc. ACM Symposium on Theory of Computing, 1978.

[12] E. Moggi. *The partial Lambda Calculus.* PhD thesis, University of Edinburgh, 1990.

[13] T. Strahm. *Theories with self-application and computational complexity.* Submitted, 2001. URL: `http://iamwww.unibe.ch/~til/publications/files/submitted/comp.pdf`