Computational Complexity and Induction for Partial Computable Functions in Type Theory

Robert L. Constable
Cornell University

Karl Crary
Carnegie Mellon University

March 2, 1999

Abstract

An adequate theory of partial computable functions should provide a basis for defining *computational complexity measures* and should justify the principle of *computational induction* for reasoning about programs on the basis of their recursive calls. There is no practical account of these notions in type theory, and consequently such concepts are not available in applications of type theory where they are greatly needed. It is also not clear how to provide a practical and adequate account in programming logics based on set theory.

This paper provides a practical theory supporting all these concepts in the setting of constructive type theories. We first introduce an extensional theory of partial computable functions in type theory. We then add support for intensional reasoning about programs by explicitly reflecting the essential properties of the underlying computation system. We use the resulting intensional reasoning tools to justify computational induction and to define computational complexity classes. Complexity classes take the form of *complexity-constrained function types.* These function types are also used in conjunction with the propositions-as-types principle to define a *resource-bounded logic* in which proofs of existence can guarantee feasibility of construction.

1   Introduction

Over the past two decades, type theory has become the formalism of choice to support programming, verification and the logical foundations of computer science. The language of types underlies modern programming languages like Java and ML, and the theory of types drives significant efforts in compilation [24, 43, 31, 34, 36, 7, 40] and semantics [8, 20, 16]. Theorem proving systems based on type theory have been used for the verification of both hardware and software, and have also been very widely used for the formalization of mathematics.

One of the major reasons type theory has enjoyed such wide successes is that it is a natural high-level language for computational mathematics and programming. However, this advantage can sometimes pose a problem because the needs of mathematics and programming can diverge. In mathematics, equality is *extensional,* where only an object's value is significant. That is, if the result of $f(a)$ is b then $f(a) = b$, and functions f and g are equal (in $A \to B$) exactly when $f(a) = g(a)$ for every a in A. In contrast, in the analysis of programs, it is often critical to consider programs *intensionally,* reasoning about their structure as well as their result value. For example, showing

1

partial correctness of recursive procedures requires reasoning inductively about the computation of recursive calls. Also, computational complexity certainly depends on a program's algorithm, and not only on its result.

So a programming logic must reconcile two needs: one, the need to treat functions *extensionally* in order to interface with mathematics and to express the most general laws of substitution; and, two, the need to treat functions *intensionally* (as algorithms) in order to state the most useful rules of program reasoning and to express complexity. There have been several attempts at this reconciliation, some of which are theoretically quite interesting [11, 42, 3, 29, 30]. Nevertheless, none of these are very practical, nor have any been implemented.

We set as our goal producing a practical and adequate account of partial functions, computational induction and computational complexity in the class of constructive type theories such as those of Martin-Löf or Girard. This means that the type theory is built on an underlying *computation system* (or programming language), and reasoning about this computation system will be central to our account. The particular type theory we have chosen in which to formalize our account is summarized in Section 2.

1.1   Elements of a Solution

Solving this problem is largely a matter of design informed by mathematical analysis. We know that some form of intensional analysis is required. It can be found by building internal *models* of computation, or, in constructive theories, by exposing parts of the underlying term structure (as in reflection [2]). The wrong choices lead to impractical and unusable theories. For example, building an internal model of a machine (say a random access machine) and a semantic map from machines to functions would express an obvious way to think about procedures, structure and complexity, but that formalization would be so heavy as to overwhelm the logic. Likewise, defining an internal semantics for the programming language component of a constructive type theory (a deep embedding) would be extremely complex.

Our judgement is that in the case of constructive type theories, the first key to a sufficient account is to bring into the type theory parts of the mechanism of the underlying computation system. Then the semantic rules connecting terms to objects (algorithms to functions) are the same as the typing rules of the type theory itself. The second key is to expose only those elements of the computation system that are needed for computational induction and complexity. The resulting extension to the type theory is lightweight but still quite expressive.

1.2   Salient Points

The first technical step is to extend the type theory with an *extensional* notion of partial functions. We take the approach introduced by Constable and Smith [11] of adding *partial types*. An object t is said to be in the partial type $\overline{T}$ exactly when its termination implies it is in T. The partial functions from A to B are denoted by $A \to \overline{B}$. However, we must extend the theory of Constable and Smith to deal with equality, which they did not consider.

The second key technical step is to add a lightweight reflection mechanism: The metatype of terms

is reflected into the type theory as the internal type *Term*.[1] This allows for intensional consideration of terms. A few other constructs are also introduced to reflect the computational behavior of terms. These intensional mechanisms allow us to define and axiomatize the type $[T]$, the subtype of *Term* that contains the *internal representations* of all terms in T.

Then the principle of computational induction is naturally expressible. This principle is based on the idea that if evaluation of t terminates, a well-founded order is induced on the terms through which it evaluates. This principle is *essentially* intensional; a term cannot be distinguished extensionally from a term to which it evaluates. In Section 4 we formalize this principle and present some examples of its uses.

Finally, using these tools we may define computational complexity classes. We define the *size* of a term as the size of its syntax tree, and its *time* as the length of its evaluation sequence to canonical form. These definitions may be used to refine the type $[T]$ of all terms in T to the type $[T]_s^t$ of all terms in T that compute within time t and space s. More interestingly, we may define complexity classes in all types using *complexity-constrained function types* such the type $A \xrightarrow{\text{ptime}} B$ of polynomial-time computable functions from A to B.

One particular significant application of this complexity machinery is to define a *resource-bounded logic* using complexity-constrained function types and the propositions-as-types principle [21]. One of the key advantages of constructive logic is that when the existence of an object is proven, that object may be constructed. However, there never been a guarantee that such a construction will be feasible to perform. By proving the existence of an object in resource-bounded logic, we guarantee that the object may be constructed feasibly [35].

2   Partial Object Type Theory

In this section we lay out the underlying type theory for the results of this paper. Our work builds on the type theory of Nuprl [9], a type theory in the style of Martin-Löf [26, 27]. We briefly summarize Nuprl in Sections 2.1. This starting theory is essentially a theory of total functions only, but it may be extended to a theory of partial computable functions by adding partial types (types containing divergent terms) and a fixpoint rule for typing recursive functions [11, 12, 42, 41, 4, 17]. We use Crary's account [17] of type theory augmented by partial functions because that account is unique in providing an equality assertion within the logic, which we require in order to support extensional reasoning.

2.1   Type Theory Preliminaries

As data types, the theory contains integers (denoted by $\mathbb{Z}$), disjoint unions (denoted by $T_1 + T_2$), lists (denoted by T *list*), dependent products (denoted by $(x{:}T_1) \times T_2$) and dependent function spaces (denoted by $(x{:}T_1) \to T_2$). As usual, when x does not appear free in T_2, we write $T_1 \times T_2$ for $(x{:}T_1) \times T_2$ and $T_1 \to T_2$ for $(x{:}T_1) \to T_2$. Types themselves are data in the theory and belong to a predicative hierarchy of universes, $\mathbb{U}_1, \mathbb{U}_2, \mathbb{U}_3$, etc. The universe $\mathbb{U}_1$ contains all types built from

---

[1] A similar approach was used by Allen, *et al.* [2], but our mechanism is much more lightweight because we do not attempt to reflect proofs as well.

	Type Formation	Introduction	Elimination
universe i	$\mathbb{U}_i$ (for $i \geq 1$)	type formation operators	
disjoint union	$T_1 + T_2$	$inj_1(e)$ $inj_2(e)$	$case(e, x_1.e_1, x_2.e_2)$
function space	$(x{:}T_1) \to T_2$	$\lambda x.e$	$e_1 e_2$
product space	$(x{:}T_1) \times T_2$	$\langle e_1, e_2 \rangle$	$\pi_1(e)$ $\pi_2(e)$
integers	$\mathbb{Z}$	$\ldots, -1, 0, 1, 2, \ldots$	assorted operations
equality	$t_1 = t_2 \ in \ T$	$\star$	
inequality	$t_1 \leq t_2$	$\star$	
set type	$\{x : T_1 \mid T_2\}$	operators for T_1	
quotient type	$x, y{:}T_1 /\!/ T_2$	operators for T_1	

Figure 1: Basic Type Theory Syntax

the base types only (*i.e.*, built without universes), and the universe $\mathbb{U}_{i+1}$ contains all types build from the base types and the universes $\mathbb{U}_1, \ldots, \mathbb{U}_i$. In particular, no universe is a member of itself.

Propositions are interpreted as types using the propositions-as-types principle [21]. This gives interpretations to the basic logical connectives and justifies using $\mathbb{U}_i$ as the type of propositions (of level i). Other assertions are interpreted using the following device: The type $n_1 \leq n_2$ is well-formed when n_1 and n_2 are integers, and either is inhabited by the term $\star$ (when, in fact, $n_1 \leq n_2$) or is empty (if $n_1 > n_2$).

Each type T comes with an intrinsic equality relation denoted by $t_1 = t_2 \in T$. Membership is also derived from this relation; we say that $t \in T$ if $t = t \in T$. The equality relation is interpreted in the logic by the type $t_1 = t_2 \ in \ T$ using the device described above. (Note that $t_1 = t_2 \in T$ is a metatheoretical assertion whereas $t_1 = t_2 \ in \ T$ is a type in the theory.) The type *Void* is then defined as $0 = 1 \ in \ \mathbb{Z}$ and the type *Unit* is defined as $0 = 0 \ in \ \mathbb{Z}$. Then the booleans ($\mathbb{B}$) are defined as *Unit* + *Unit* and *true*, *false* and *if-then-else* constructs are defined in the obvious manner.

Propositions may be used to construct new types using the set type and quotient type constructors. The set type $\{x : A \mid P\}$ contains all elements t of A such that $P[t/x]$ is inhabited (where $E[e/x]$ denotes the capture-avoiding substitution of e for x in E). We use the set type to define the natural numbers ($\mathbb{N}$) as $\{n : \mathbb{Z} \mid 0 \leq n\}$, the positive integers ($\mathbb{Z}^+$) as $\{n : \mathbb{Z} \mid 1 \leq n\}$, and the integer subranges ($[m \cdots k]$) as $\{n : \mathbb{Z} \mid m \leq n \wedge n \leq k\}$. The quotient type $x, y{:}A /\!/ P$ contains all elements of A, but t_1 and t_2 are equal in $x, y{:}A /\!/ P$ if $P[t_1, t_2/x, y]$ is inhabited. For such a type to be well-formed, the equality resulting from P must be coarser than the equality on A.

2.2 Computation

Underlying our type theory is a computation system that is summarized in Appendix B. The computation system is defined by a small-step evaluation relation (denoted by $t_1 \mapsto t_2$), and a set of canonical terms. (Of the terms introduced so far, the canonical terms are those appearing in the first and second columns of Figure 1.) Two properties of the computation system are that it is deterministic (*i.e.*, if $t \mapsto t_1$ and $t \mapsto t_2$ then $t_1 =_\alpha t_2$) and that canonical forms are terminal (*i.e.*,

if $canon(t)$ then $t \not\mapsto t'$).

If $t \mapsto^* t'$ and $canon(t')$ then we say that t converges (abbreviated $t\!\downarrow$) and t converges to t' (abbreviated $t \Downarrow t'$). Note that if $t \Downarrow t_1$ and $t \Downarrow t_2$ then $t_1 =_\alpha t_2$ and that if $t \Downarrow t'$ then $t' \Downarrow t'$. We say that a term t diverges (abbreviated $t\!\uparrow$) if it does not converge.

The equality on types is constructed to obey an important property with respect to computation, that equality is value respecting: if $t \in T$ and $t \mapsto t'$ then $t = t' \in T$. As discussed in the introduction, this property is critical to mathematical uses of the theory, but it is this same property that makes intensional reasoning challenging: within the type theory, a term and its result value are equal, as are any two terms with the same result value.

2.3   Partial Types

Functions (and objects in general) may be defined recursively in the basic type theory using the *fix* operator. However, all the types of the basic type theory are total (*i.e.,* contain only convergent elements), so to show that a recursively defined object is well-typed, it is necessary to show that it converges. Consequently, the basic theory is incapable of reasoning about possibly partial functions. To make the theory capable of reasoning about partial functions, we will add four new primitive types: a partial type constructor, and types asserting convergence, admissibility and totality.

The *partial type* $\overline{T}$ is like a "lifted" version of T; it contains all the members of T as well as all divergent terms. Partial functions from A to B may then be given the type $A \rightarrow \overline{B}$. For reasons that will become clear shortly, it is important that $\overline{T}$ never equate any convergent and divergent terms. Thus we say that two terms are equal in $\overline{T}$ if they both diverge, or if they both converge and are equal in T. However, we desire that T be a subtype of $\overline{T}$ (*i.e.,* if $t_1 = t_2 \in T$ then $t_1 = t_2 \in \overline{T}$), so in order for $\overline{T}$ to be allowed as a valid type, we must require that T also never equate any convergent and divergent terms. Instead of this condition, we require the stronger condition that T contain only convergent elements, which seems to work more nicely in practice.

We reason about convergence in the theory using the type $t \ in!\ T$, which is well-formed when $t \in \overline{T}$ and inhabited when $t\!\downarrow$. We wish to say that the types $t \ in!\ T$ and $t' \ in!\ T$ are equal when $t = t' \in \overline{T}$, and it is for this reason that we cannot permit $\overline{T}$ to equate convergent and divergent terms.

The well-formedness conditions of the partial type and the convergence type contain an apparent circularity: Well-formedness of the convergence type $t \ in!\ T$ requires that $t \ in\ \overline{T}$ and the well-formedness of $\overline{T}$ requires that every member of T converge. To break this loop, we use a primitive totality predicate ($T\ total$) as the condition for partial type well-formedness within the logic.

Recursively defined objects may be typed directly (without showing they converge) using the fixpoint principle:

$$t \ in\ (\overline{T} \rightarrow \overline{T}) \Rightarrow fix(t)\ in\ \overline{T}$$

However, the fixpoint principle is not valid for all types (see Smith [42] for an example of such a type). Types for which the fixpoint principle is valid are called *admissible*. Admissibility is indicated within the logic by the type $T\ admiss$. Crary [15, 17] shows that a wide class of types are admissible, including all types used in conventional programming languages.

5

3 Intensional Reasoning

To lay the groundwork that makes intensional reasoning possible in our type theory, we begin by being more specific about the structure of terms in Section 3.1. This structure is quite general, allowing term constructors with arbitrarily many subterms and binding positions, and therefore makes few restrictions on the expressiveness of the theory. Moreover, the results of this paper should easily apply to different term structures, but we desire a concrete framework in which to work. In Section 3.2 we present operators and notation for intensional consideration of terms. Section 3.3 then presents the primitive type-theoretic constructs necessary for intensional reasoning.

3.1 Uniform Term Syntax

Terms are constructed using a set Ω of *operators*, ranged over by the metavariable ω, and a function *arity* mapping each operator to a finite (possibly empty) list of natural numbers. The arity function will be used to indicate the number of subterms of an operator and the number of bound variables for each subterm. We also take as given an infinite set of variables.

The set of terms is then defined inductively as follows:

- Every variable is a term.

- If ω is an operator then ω^* is a term, serving as the name of ω.

- If $arity(\omega) = (k_1, \ldots, k_n)$ and $t_1, \ldots, t_n$ are terms, then $\omega(x_{11} \cdots x_{1k_1}.t_1, \ldots, x_{n1} \cdots x_{nk_n}.t_n)$ is a (compound) term, where x_{ij} is a variable for $1 \leq i \leq n$ and $1 \leq j \leq k_i$.

The variables $x_{i1}, \ldots, x_{ik_i}$ preceding each subterm are binding occurrences. Thus, in the term $lambda(x.add(x, y))$, the variable x is bound and the variable y is free. As usual, terms that differ only by the alpha-variation of bound variables are considered identical.

A subset of the operators in Ω are designated as *canonical* operators. A term is considered canonical either if it is a compound term constructed with a canonical operator at the top, or if it is the name ω^* of an operator (canonical or not).

Although it will often be useful to present terms in the uniform term syntax just described, we will continue usually to use more conventional notation such as that used in Section 2. These conventional notations should be viewed as abbreviations for formal terms in the uniform syntax above. For example, $\lambda x.t$ is shorthand for $lambda(x.t)$ where $arity(lambda) = (1)$.

3.2 Intensional Representations

In order to analyze terms intensionally, we must have internal representations of terms. As discussed in the introduction, the representation of a term cannot be the term itself, as appealing as that might be. In particular, the representation of a term must be a canonical form, even when the term being represented is not.

$$\begin{aligned}
\lceil t \rceil &\overset{\text{def}}{=} \lceil t \rceil_\epsilon \\
\lceil x_i \rceil_{x_n,\ldots,x_1} &\overset{\text{def}}{=} var_i \qquad\qquad (\text{for } 1 \le i \le n) \\
\lceil \omega^* \rceil_{x_n,\ldots,x_1} &\overset{\text{def}}{=} opname(\omega^*) \\
\lceil \omega(y_{k_1}\cdots y_1.t_1,\ldots,y_{k_n}\cdots y_1.t_n) \rceil_{x_m,\ldots,x_1} &\overset{\text{def}}{=} \langle\!\langle \omega^*, \lceil t_1 \rceil_{x_m,\ldots,x_1,y_{k_1},\ldots,y_1}, \\
&\qquad\qquad \vdots \\
&\qquad\quad \lceil t_n \rceil_{x_m,\ldots,x_1,y_{k_n},\ldots,y_1} \rangle\!\rangle \\
&\qquad (\text{for } arity(\omega) = (k_1,\ldots,k_n)) \\
\langle\!\langle t_1,\ldots,t_n \rangle\!\rangle &\overset{\text{def}}{=} inj_1(\langle t_1, \langle t_2, \langle \cdots \langle t_n, \star \rangle \cdots \rangle\rangle\rangle) \\
var_i &\overset{\text{def}}{=} inj_2(inj_1(i)) \\
opname(t) &\overset{\text{def}}{=} inj_2(inj_2(t))
\end{aligned}$$

Figure 2: Term Representations

The representation of any term t is denoted by $\lceil t \rceil$. If If ω is an n-place operator with no binding occurrences then the term $\omega(t_1,\ldots,t_n)$ is represented by $\langle\!\langle \omega^*, \lceil t_1 \rceil, \ldots, \lceil t_n \rceil \rangle\!\rangle$. To avoid the syntactic overhead of formalizing variables, terms with binding structure are representing using de Bruijn indices [19]; for $i > 0$, var_i refers to the ith enclosing binding occurrence, counting from one. For example, $\lceil \lambda x.\lambda y.x \rceil = \langle\!\langle lambda^*, \langle\!\langle lambda^*, var_2 \rangle\!\rangle \rangle\!\rangle$. This is summarized formally in Figure 2.

Finally, it will prove convenient to have some notation like the quasiquote mechanism of Scheme for the representations of terms with holes to be filled by ordinary terms. We will denote these with concentric boxes: $\boxed{t}$ stands for the representation of t except that inner boxes within t denote holes and expressions within those holes are left unchanged. For example, if t is a term, $\boxed{\lambda x.\langle \boxed{t}, 0 \rangle}$ denotes $\langle\!\langle lambda^*, \langle\!\langle pair^*, t, number_0 \rangle\!\rangle \rangle\!\rangle$.

3.3 Type Theory of Terms

We are now ready to introduce the principal devices for intensional reasoning in type theory. These include a type of intensional terms, an operator for converting intensional terms to their denotations, and types asserting intensional type membership, evaluation, canonicalism, and computational inducement:

- The type $Term_i$ contains the representations of all terms with up to i free variables (that is, de Bruijn indices extending up to i binding positions out of the term). (*E.g.*, var_i is in $Term_j$ when $i \le j$.) Expressions are equal in $Term_i$ if they represent alpha-equal terms. Note that the representations of all terms are canonical (since *pair* is a canonical operator), even if the terms themselves are not, and therefore $Term_i$ is a total type. The type $Term$ is defined to be the type $Term_0$ of closed term representations.

- Terms are related to their representations by a set of operators ref_n (for natural numbers n), which compute the meaning of intensional term representations. The operator ref_n is used when the representing term t may contain up to n free variables: $ref_n(t, e_1,\ldots,e_n)$ computes the denotation of t where e_i is substituted for the ith enclosing free variable of t. For example, $ref_2(var_1, t_1, t_2) \mapsto t_1$. When n is small, we will write $[\![t]\!]_{e_1,\ldots,e_n}$ for $ref_n(t, e_1,\ldots,e_n)$. Note

7

that, unlike $\lceil \cdot \rceil$, ref_n is an operator and thus may be used within the theory (and indeed will be, extensively).

- Using ref, we may form the assertion that an intensional term t represents a member of T in the natural manner as $[\![t]\!]$ in T. For technical reasons,[2] the theory also requires a primitive type asserting that a term represents a member of a type. This type is t $[in]$ T, asserting that $[\![t]\!] \in T$.

  Using this type, we may define an essential refinement of the type $Term$: the type $[T]$, which contains representations of all terms of type T.

$$[T] \stackrel{\text{def}}{=} \{x : Term \mid x \ [in] \ T\}$$

  Types $[T]$, along with the $[\![\cdot]\!]$ operator, will serve as the principal interface between the extensional and intensional aspects of our type theory. When we wish to consider an unknown term x both intensionally and extensionally and x extensionally is to have type T, we will give x the type $[T]$ and then use x bare in intensional contexts but use it as $[\![x]\!]$ in extensional contexts.

- To reason about computation, we need types representing the assertions that one term evaluates to another and that a term is canonical. The type t_1 $evalto$ t_2 is inhabited by the term $\star$ when $[\![t_1]\!] \mapsto [\![t_2]\!]$ and is empty otherwise. Similarly, the type t $canonical$ is inhabited (by $\star$) exactly when $[\![t]\!]$ is canonical.

- We also need to reason about the related concept of computational inducement. If $e_1 \mapsto e_2$ then the evaluation of e_1 certainly induces the evaluation of e_2. However, depending upon the structure of e_1, the evaluation of other terms may be induced as well. For example, the evaluation of fa induces the evaluation of f. When the evaluation of e_1 to canonical form contains the evaluation of e_2 we say that e_1 $induces$ $evaluation$ of e_2. This is represented by the type t_1 $induces$ t_2, which is inhabited by the term $\star$ when $[\![t_1]\!]$ induces evaluation of $[\![t_2]\!]$ and is empty otherwise. When a term t halts, inducement defines a well-founded relation on the terms induced by t; we will use that relation for computational induction in the next section.

This type theory is formalized in Appendices A and B, which present the type theory's inference rules and operational semantics. In addition to the primitive construct discussed above, we will also need a few functions operating on intensional representations:

- A substitution function $subst \in Term_1 \to Term \to Term$ such that $subst\, e_1\, e_2$ substitutes e_2 for var_1 in e_1 (and shifts other variables down appropriately). We will often abbreviate $subst\, e_1\, e_2$ by $e_1[e_2/1]$.

- Effective versions of the evaluation and canonicalism assertions, that is, a function $iscanon \in Term \to \mathbb{B}$ that determines whether a term is canonical and a function $next \in Term \to (Term + Unit)$ that determines the next term that a term evaluates to, if it is not canonical or stuck.

---

[2]In Martin-Löf style type theories, the membership type t in T is well-formed only when t is a member of T. Hence, the assertion $[\![t]\!]$ in T is well-formed only when it is true, and thus it is useless as the antecedent of an implication. We will need to form propositions that are conditional on an intensional term's denotation belonging to a type, so we need a type that is inhabited if and only if the above type is, but that has more liberal conditions on well-formedness. From t $[in]$ T we may deduce that $[\![t]\!]$ in T and vice versa, but t $[in]$ T is well-formed whenever t is a member of $Term$ (and T is a type).

Note that the last three functions (*subst, iscanon,* and *next*) can be written in the underlying type theory and need not be made primitive. Moreover, they will still be programmable in any reasonable extension of our computation system: the programmability of *subst* is inherent in the syntax of terms and their intensional represenations, and the programmability of *iscanon* and *next* amount to the constructability of a universal machine, which is usually part of the definition of a reasonable computation system.

4 Computational Induction

With a type theory capable of intensional reasoning in hand, we are ready to look at some applications of intensional reasoning. The first *computational induction,* a useful principle of induction that is not valid in extensional type theories. Computational induction is based on the inducement order relation: if the evaluation of a term t converges, inducement is a well-founded order over those terms induced by t. (Recall that t induces t' when the evaluation of t includes the evaluation of t'.)

Computational induction has general applicability for reasoning about programs since it corresponds to the intuitive mechanism of reasoning about recursive programs by preconditions and postconditions, but it is particularly useful for proving *partial* correctness of programs. Other induction principles are inadequate for such proofs; if one could find a well-founded order on the inputs of a recursive function, one could show total correctness instead.

The principle of computational induction is as follows:

$$(\forall e{:}Term.\,(\forall e'{:}Term.\;e\;induces\;e' \Rightarrow P[e']) \Rightarrow P[e]) \wedge [\![t]\!]\;in!\;T \Rightarrow P[t]$$

In words, if we may show $P[e]$ from the assumption of $P[e']$ for every e' induced by e, and if t halts, then $P[t]$.

Note that, unlike the fixpoint principle, there is no admissibility restriction on computational induction. This is because, like most induction principles but unlike fixpoint induction, computational induction is based on a well-founded order. The generality of computational induction makes it a useful tool for reasoning about partial function in circumstances where fixpoint induction is disallowed.

Note also that since inducement is an intrinsically intensional property, the computational induction principle demands the intensional reasoning structure developed in Section 3 to state. Indeed, if the principle were expressed in extensional terms, it would be inconsistent: If e evaluates to e', then e also induces e', but because equality is value respecting e will be equal to e' is any types to which they belong. Thus, if computational induction were expressed extensionally (that is, if it repected equality), then the first condition would always be satisfied, allowing any property to be proven of any terminating term.

We will now examine some examples of the use of compuational induction:

Example 1 Consider the "$3n + 1$" function:

$$F = fix(\lambda f.\lambda n.if\ n = 1\ then\ 0$$
$$else\ if\ n\ \mathrm{mod}\ 2 = 0\ then\ 1 + f(n/2)$$
$$else\ f(3n + 1))$$

This program is not known to halt for all positive integers, but it is easy to see that if $F(n){\downarrow}$ then $\log_2 n \leq F(n)$. We may show this by computational induction using the proposition:

$$P[e]\ =\ \forall n{:}[\mathbb{Z}^+].\,(e = \boxed{F(\boxed{n})}\ in\ Term) \Rightarrow \log_2 [\![n]\!] \leq [\![e]\!]$$

Proof

Let $e\ in\ Term$ be arbitrary. Suppose $P[e']$ for all e' induced by e and suppose $e = \boxed{F(\boxed{n})}$ in $Term$ (for some n in $[\mathbb{Z}^+]$). By assumption, $[\![n]\!]$ in $\mathbb{Z}^+$. If $[\![n]\!] = 1$ then $[\![e]\!] = 0$. Suppose $[\![n]\!]\ \mathrm{mod}\ 2 = 0$. Then $\boxed{F(\boxed{n})}$ $evalto$ $\boxed{1 + F(\boxed{n}/2)}$, and thus e induces $\boxed{F(\boxed{n}/2)}$. By induction $\log_2([\![n]\!]/2) \leq F([\![n]\!]/2)$, so $\log_2 [\![n]\!] = 1 + \log_2([\![n]\!]/2) \leq 1 + F([\![n]\!]/2) = F([\![n]\!])$.

The remaining case is similar. Suppose $[\![n]\!]\ \mathrm{mod}\ 2 = 1$. Then $\boxed{F(\boxed{n})}$ $evalto$ $\boxed{F(3\boxed{n}+1)}$, and thus e induces $\boxed{F(3\boxed{n}+1)}$. By induction $\log_2(3[\![n]\!] + 1) \leq F(3[\![n]\!] + 1)$, so $\log_2 [\![n]\!] \leq \log_2(3[\![n]\!]+1) \leq F(3[\![n]\!]+1) = F[\![n]\!]$. By computational induction we conclude $[\![t]\!]\ in!\ \mathbb{Z} \Rightarrow P[t]$. It follows that $\forall n{:}[\mathbb{Z}^+].\,F[\![n]\!]\ in!\ \mathbb{Z} \Rightarrow \log_2 [\![n]\!] \leq F[\![n]\!]$, as desired.

Example 2 Computational induction is also useful for showing conditional termination. Consider the function $time$ that computes the length of the reduction sequence of a term:

$$time\ \stackrel{\mathrm{def}}{=}\ fix(\lambda f.\lambda t.case(next(t), t'.1 + f(t'), x.0))$$

We may use computational induction to show that $time(t)\ in!\ \mathbb{N}$ whenever $[\![t]\!]\ in!\ T$ (for any T).[3] This is done using the proposition $P[e] = time(e)\ in!\ \mathbb{N}$ and the observation that if $next(e) = inj_1(e')$ then e induces e' (since $e\ evalto\ e'$).

5 Computational Complexity and Resource-Bounded Logic

A second application of intensional reasoning lies in computational complexity. The evaluation relation on $Term$ provides the basis for defining computational complexity measures such as time and space. These measures allow us to express traditional results about complexity classes as well as recent results concerning complexity in higher types [6, 37, 22, 13, 14, 38, 39, 23, 44]. The basic measure of time is the number of evaluation steps to canonical form. We have already defined (in the previous section) a function counting the evaluation steps of a term, so we already have a notion of running time.[4] However, it will prove more convenient to state that notion in predicate

---

[3]The fixpoint principle suffices to show $\forall t{:}Term.\,time(t) \in \overline{\mathbb{N}}$.

[4]Since the underlying computation system allows $(\lambda x.b)a$ to evaluate in one step to $b[a/x]$ (an action impossible in constant time on real machines), this measure of running time will not always correspond to measured derived for more realistic machine models. Fortunately, the theory we present does not depend greatly on the actual computation

form where $Term(e, t)$ states that e runs within time t:

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

We may define a notion of space in a similar manner. First, we may easily define a function *size* with type $Term \to \mathbb{N}$ which computes the number of operators in a term. Then we define the predicate $Space(e, s)$, that states that e runs in space at most s:

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

Using these, we may define the resource-indexed type $[T]_s^t$ of terms that evaluate (to a member of T) within time t and space s:

$$[T]_s^t \ \stackrel{\text{def}}{=} \ \{e : [T] \mid Time(e, t) \wedge Space(e, s)\}$$

One interesting application of the resource-indexed types is to define types like Parikh's feasible numbers [35], numbers that may be computed in a "reasonable" time.

5.1 Complexity Classes

With time complexity measures defined above, we may define complexity classes of functions. Complexity classes are expressed as function types whose members are required to fit within complexity constraints. We call such type *complexity-constrained* function types. For example, the quadratic time, polynomial time, and polynomial space computable functions may be defined in a straightforward manner:

$$(x{:}A) \xrightarrow{\text{quad}} B \ \stackrel{\text{def}}{=} \ \{f : [(x{:}A) \to B] \mid \exists c{:}\mathbb{N}.\, \forall a{:}[A].\, Time(\langle\!\langle app^*, f, a \rangle\!\rangle, c \cdot size(a)^2)\}$$
$$(x{:}A) \xrightarrow{\text{ptime}} B \ \stackrel{\text{def}}{=} \ \{f : [(x{:}A) \to B] \mid \exists c, c'{:}\mathbb{N}.\, \forall a{:}[A].\, Time(\langle\!\langle app^*, f, a \rangle\!\rangle, c \cdot size(a)^{c'})\}$$
$$(x{:}A) \xrightarrow{\text{pspace}} B \ \stackrel{\text{def}}{=} \ \{f : [(x{:}A) \to B] \mid \exists c, c'{:}\mathbb{N}.\, \forall a{:}[A].\, Space(\langle\!\langle app^*, f, a \rangle\!\rangle, c \cdot size(a)^{c'})\}$$

While this technique provides complexity classes in a theory rich enough to express all mathematics, notions of complexity over *higher types* rely on *oracle time* complexity [28, 13, 14], where evaluation of a distinguished argument is not counted toward evaluation time, but the processing of its result is. Supporting such notions in this framework requires a mechanism for marking the distinguished argument throughout evaluation so that it may always be identified and discounted. Such a mechanism allows the definition of higher-order polynomial time computable functions [10, 28, 13, 39]. It is not difficult to produce such a mechanism in this framework, but we will not discuss it here due to space limitations.

---

system used; many computation systems could validate our axioms, and we have chosen the one used in this paper only for its simplicity. A computation system based on explicit substitutions [1, 18] can also easily be used, and leads to a more realistic measure of running time. Still more realistic computation systems have also been proposed, such as the λ_{gc} of Morrisett *et al.* [32, 33], but we have not explored how easily our theory could be adapted to such systems.

5.2 Resource-Bounded Logic

One of the advantages of constructive logic is that when the existence of an object is proven, that object may be constructed. Many theorem provers based on constructive logic [9, 5, 25] allow the automatic extraction of programs from proofs. However, there is no guarantee that such programs may feasibly be executed. This has been a serious problem in practice, as well as in principle.

Using the complexity-constrained functions, we may define a resource-bounded logic that solves this problem. Under the propositions-as-types principle, the universal statement $\forall x{:}A.B$ corresponds to the function space $(x{:}A) \to B$. By using the complexity-constrained function space instead, we obtain a *resource-bounded* universal quantifier. For example, let us denote the quantifier corresponding to the polynomial-time computable functions by $\forall_{(ptime)} x{:}A.B$. By proving the statement $\forall_{(ptime)} x{:}A. \exists y{:}B. P(x, y)$, we guarantee that the appropriate y may actually be feasibly computed from a given x.

6 Conclusion

We have presented a type theory that reconciles extensional and intensional reasoning, providing the first account of computational complexity that does not incur heavy mechanisms such as reflection or deep embeddings. The constructs we have presented may be implemented in terms of the theory of reflection presented by Allen *et al.* [2], and indeed our theory meshes nicely with reflection. However, our theory is much more lightweight than the theory of reflection, which internalizes sequents, proofs and programs that construct proofs, so it can be profitable to justify the constructs of our theory as primitives instead. We argue, then, that our theory occupies an attractive middle position, expressive enough for powerful reasoning (such as complexity analysis), and consistent with richer theories such as reflection, but simple enough to be used practically.

A Inference Rules

A complete set of inference rules for the type theory with extensional partial functions is given in Crary [17]. In this appendix, we present those additional rules needed to reason intensionally:

Terms

$$\frac{H \vdash j = j' \ in \ \mathbb{N}}{H \vdash \ Term_j = Term_{j'} \ in \ \mathbb{U}_i} \qquad \frac{}{H \vdash Op = Op \ in \ \mathbb{U}_i} \qquad \frac{}{H \vdash \omega^* = \omega^* \ in \ Op}$$

$$\frac{H \vdash t = t' \ in \ Op}{H \vdash ar(t) = ar(t') \ in \ \mathbb{N} \ list} \qquad \frac{}{H \vdash ar(\omega^*) = (k_1 :: \cdots :: k_n :: nil) \ in \ \mathbb{N} \ list} \ (arity(\omega^*) = (k_1, \ldots, k_n))$$

$$\frac{H \vdash e = e' \ in \ ((o{:}Op) \times fold(ar(o), i\, X.\, Term_{i+j} \times X, Unit)) + [1 \cdots i] + Op}{H \vdash e = e' \ in \ Term_j}$$

12

$$\frac{H \vdash e = e' \ in \ Term_j}{H \vdash e = e' \ in \ ((o{:}Op) \times fold(ar(o), i\, X.\, Term_{i+j} \times X,\, Unit)) + [1 \cdots i] + Op}$$

Intensional Membership

$$\frac{H \vdash t = t' \ in \ Term \quad H \vdash T = T' \ in \ \mathbb{U}_i}{H \vdash (t \ [in] \ T) = (t' \ [in] \ T') \ in \ \mathbb{U}_i} \qquad \frac{H \vdash [\![t]\!] \ in \ T}{H \vdash t \ [in] \ T} \qquad \frac{H \vdash t \ [in] \ T}{H \vdash [\![t]\!] \ in \ T}$$

Evaluation, Canonicalism, and Inducement

$$\frac{H \vdash t_1 = t_1' \ in \ Term \quad H \vdash t_2 = t_2' \ in \ Term}{H \vdash (t_1 \ evalto \ t_2) = (t_1' \ evalto \ t_2') \ in \ \mathbb{U}_i} \qquad \frac{H \vdash t = t' \ in \ Term}{H \vdash (t \ canonical) = (t' \ canonical) \ in \ \mathbb{U}_i}$$

$$\frac{H \vdash t_1 = t_1' \ in \ Term \quad H \vdash t_2 = t_2' \ in \ Term}{H \vdash (t_1 \ induces \ t_2) = (t_1' \ induces \ t_2') \ in \ \mathbb{U}_i} \qquad \frac{H \vdash t_1 \ induces \ t_2}{H \vdash \neg(t_1 \ canonical)} \qquad \frac{H \vdash t_1 \ evalto \ t_2}{H \vdash t_1 \ induces \ t_2}$$

$$\frac{H \vdash t_1 \ evalto \ t_2 \quad H \vdash [\![t_1]\!] \ in \ T}{H \vdash [\![t_1]\!] = [\![t_2]\!] \ in \ T} \qquad \frac{H \vdash t \ canonical \quad H \vdash [\![t]\!] \ in \ T \quad H \vdash \overline{T} \ in \ \mathbb{U}_i}{H \vdash [\![t]\!] \ in! \ T}$$

$$\frac{H \vdash [\![t]\!] \ in! \ T \quad H, e{:}Term, (\forall e'{:}Term.\ e \ induces \ e' \Rightarrow P[e'/x]) \vdash P[e/x]}{H \vdash P[t/x]}$$

Sample Denotation Rules

$$\frac{H, y : [A], z : [A], w : ([\![y]\!] = [\![z]\!] \ in \ A) \vdash [\![b[y/1]]\!] = [\![b[z/1]]\!] \ in \ B[[\![y]\!]/x]}{H \vdash [\![\ \boxed{\lambda x.\boxed{b}}\ ]\!] = \lambda x.[\![b]\!]_x \ in \ (x : A) \to B}$$

$$\frac{H \vdash e_1 \ [in] \ (x : A) \to B \quad H \vdash e_2 \ [in] \ A}{H \vdash [\![\ \boxed{\boxed{e_1}\ \boxed{e_2}}\ ]\!] = [\![e_1]\!][\![e_2]\!] : B[[\![e_1]\!]/x]}$$

Sample Evaluation and Inducement Rules

$$\frac{H \vdash e \ in \ Term_1}{H \vdash \boxed{\lambda x.\boxed{e}} \ canonical} \qquad \frac{H \vdash f \ evalto \ f' \quad H \vdash e \ in \ Term}{H \vdash \boxed{\boxed{f}\ \boxed{e}} \ evalto \ \boxed{\boxed{f'}\ \boxed{e}}}$$

$$\frac{H \vdash e \ in \ Term_1 \quad H \vdash t \ in \ Term}{H \vdash \boxed{(\lambda x.\boxed{e})\boxed{t}} \ evalto \ e[t/1]} \qquad \frac{H \vdash f \ in \ Term \quad H \vdash e \ in \ Term}{H \vdash \boxed{\boxed{f}\ \boxed{e}} \ induces \ f}$$

Lists

$$\frac{H \vdash T = T' \ in \ \mathbb{U}_i}{H \vdash T \ list = T' \ list \ in \ \mathbb{U}_i} \qquad \frac{H \vdash T \ in \ \mathbb{U}_i}{H \vdash nil = nil \ in \ T \ list} \qquad \frac{H \vdash h = h' \ in \ T \quad H \vdash t = t' \ in \ T \ list}{H \vdash h :: t = h' :: t' \ in \ T \ list}$$

$$\frac{H \vdash e_1 = e'_1 \ in \ T \ list \quad H, x : T, y : S \vdash e_2 = e'_2 \ in \ S \quad H \vdash e_3 = e'_3 \ in \ S}{H \vdash fold(e_1, x \ y.e_2, e_3) = fold(e'_1, x \ y.e'_2, e'_3) \ in \ S}$$

$$\frac{H \vdash e' \ in \ S}{H \vdash fold(nil, x \ y.e, e') = e' \ in \ S} \qquad \frac{H \vdash h \ in \ T \quad H \vdash t \ in \ T \ list \quad H, x : T, y : S \vdash e \ in \ S \quad H \vdash e' \ in \ S}{H \vdash fold(h :: t, x \ y.e, e') = e[h, fold(t, x \ y.e, e')/x, y] \ in \ S}$$

B   Operational Semantics

$$\frac{f \mapsto f'}{f \ e \mapsto f' \ e} \qquad \frac{}{(\lambda x.e)t \mapsto e[t/x]} \qquad \frac{t \mapsto t'}{\pi_i(t) \mapsto \pi_i(t')} \qquad \frac{}{\pi_i(\langle t_1, t_2 \rangle) \mapsto t_i}$$

$$\frac{t \mapsto t'}{case(t, x.e_1, x.e_2) \mapsto case(t', x.e_1, x.e_2)} \qquad \frac{}{case(inj_i(t), x.e_1, x.e_2) \mapsto e_i[t/x]}$$

$$\frac{e_1 \mapsto e'_1}{fold(e_1, x \ y.e_2, e_3) \mapsto fold(e'_1, x \ y.e_2, e_3)} \qquad \frac{}{fold(nil, x \ y.e, e') \mapsto e'}$$

$$\frac{}{fold(h :: t, x \ y.e, e') \mapsto e[h, fold(t, x \ y.e, e')/x, y]} \qquad \frac{}{fix(f) \mapsto f \ fix(f)}$$

$$\frac{t \mapsto t'}{ref_n(t, e_1, \ldots, e_n) \mapsto ref_n(t', e_1, \ldots, e_n)}$$

$$\frac{t \mapsto t'}{ref_n(inj_i(t), e_1, \ldots, e_n) \mapsto ref_n(inj_i(t'), e_1, \ldots, e_n)}$$

$$\frac{t_1 \mapsto t'_1}{ref_n(inj_1(\langle t_1, t_2 \rangle), e_1, \ldots, e_n) \mapsto ref_n(inj_1(\langle t'_1, t_2 \rangle), e_1, \ldots, e_n)}$$

$$\frac{arity(\omega) = (k_1, \ldots, k_m)}{\begin{aligned} ref_n(inj_1(\langle \omega^*, t \rangle), e_1, \ldots, e_n) \mapsto \quad & \omega(x_{k_1} \cdots x_1.ref_{k_1+n}(\pi_1(t), x_1, \ldots, x_{k_1}, e_1, \ldots, e_n), \\ & \vdots \\ & x_{k_m} \cdots x_1.ref_{k_m+n}(\pi_1(\underbrace{\pi_2(\cdots(\pi_2}_{m-1 \ \text{times}}(t))\cdots)), x_1, \ldots, x_{k_1}, e_1, \ldots, e_n)) \end{aligned}}$$

$$\frac{t \mapsto t'}{ref_n(inj_2(inj_i(t)), e_1, \ldots, e_n) \mapsto ref_n(inj_2(inj_i(t')), e_1, \ldots, e_n)}$$

$$\frac{}{ref_n(inj_2(inj_1(i)), e_1, \ldots, e_n) \mapsto e_i} \ (1 \le i \le n) \qquad \frac{}{ref_n(inj_2(inj_2(\omega^*)), e_1, \ldots, e_n) \mapsto \omega^*}$$

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Symposium on Logic in Computer Science*, pages 95–197. IEEE, June 1990.

[3] P. Audebaud. Partial objects in the calculus of constructions. In *Proc. of Sixth Symp. on Logic in Comp. Sci.*, pages 86–95, IEEE, Vrije University, Amsterdam, The Netherlands, 1991.

[4] Philippe Audebaud. Partial objects in the calculus of constructions. In *Sixth IEEE Symposium of Logic in Computer Science*, pages 86–95, Amsterdam, July 1991.

[5] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual*. INRIA-Rocquencourt, CNRS and ENS Lyon, 1996.

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

[7] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit (version 1). Technical Report 93/14, Department of Computer Science, University of Copenhagen, 1993.

[8] L. Cardelli and P. Longo. A semantic basis for Quest. In *Journal of Functional Programming*, pages 1:417–458, 1991.

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

[10] Robert L. Constable. Type two computational complexity. In *Proceedings of the 5th Annual ACM Symposium on the Theory of Computing*, pages 108–121, 1973.

[11] Robert L. Constable and Scott Fraser Smith. Partial objects in constructive type theory. In *Second IEEE Symposium of Logic in Computer Science*, pages 183–193, Ithaca, New York, June 1987.

[12] Robert L. Constable and Scott Fraser Smith. Computational foundations of basic recursive function theory. In *Third IEEE Symposium of Logic in Computer Science*, pages 360–371, Edinburgh, Scotland, July 1988.

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

[14] S. A. Cook and B. M. Kapron. A new characterization of Mehlhorn's polynomial time functionals. *FOCS*, 1991.

[15] Karl Crary. Admissibility of fixpoint induction over partial types. In *Fifteenth International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Computer Science*, pages 270–285, Lindau, Germany, July 1998. Springer-Verlag. Extended version published as CMU technical report CMU-CS-98-164.

[16] Karl Crary. Programming language semantics in foundational type theory. In *International Conference on Programming Concepts and Methods*, Shelter Island, New York, 1998. Chapman & Hall. Extended version published as Cornell University technical report TR98-1666.

[17] Karl Crary. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, August 1998.

[18] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, March 1996.

[19] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[20] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In *Milner Festschrifft*. The MIT Press, 1998. To appear.

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

[22] N. Immerman. Languages which capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.

[23] Daniel 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] Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–188, 1992.

[25] Lena Magnusson. *The Implementation of ALF—A Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Göteborg University, Göteborg Sweden, 1994.

[26] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Proceedings of the Logic Colloquium, 1973*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975.

[27] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science*, volume 104 of *Studies in Logic and the Foundations of Mathematics*, pages 153–175. North-Holland, 1982.

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

[29] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93, 1991.

[30] Eugenio Moggi. A general semantics for evaluation logic. In *Proceedings of Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 353–362, July 1994.

[31] Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, December 1995.

[32] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *1995 Conference on Functional Programming Languages and Computer Architecture*, 1995.

[33] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. Technical Report CMU-CS-96-176, Carnegie Mellon University, School of Computer Science, September 1996.

[34] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Twenty-Fifth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, San Diego, January 1998. To appear.

[35] R. Parikh. Existence and feasibility in arithmetic. *Jour. Assoc. Symbolic Logic*, 36:494–508, 1971.

[36] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, July 1993.

[37] A. Seth. There is no recursive axiomatization for feasible functionals of type 2. In *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, 1992.

[38] A. Seth. Some desirable conditions for feasible functionals of type 2. In *Eighth IEEE Symposium of Logic in Computer Science*, pages 320–331, Montreal, June 1993.

[39] A. Seth. Turing machine characterizations of feasible functionals of all finite types. In *Proceedings of MSI Workshop on Feasible Mathematics*, 1994.

[40] Zhong Shao and Andrew Appel. A type-based compiler for Standard ML. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–129, La Jolla, June 1995.

[41] Scott F. Smith. Hybrid partial-total type theory. *International Journal of Foundations of Computer Science*, 6:235–263, 1995.

[42] Scott Fraser Smith. *Partial Objects in Type Theory*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, January 1989.

[43] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, May 1996.

[44] K. Weihrauch and Ch. Kreitz. Type 2 computational complexity of functions on Cantor's space. *Theoretical Computer Science*, 82:1–18, 1991.