# A Linguistic Characterization of Bounded Oracle Computation and Probabilistic Polynomial Time

J. Mitchell * M. Mitchell † A. Scedrov ‡
Stanford University University of Pennsylvania
{mitchell,mmitchel}@cs.stanford.edu scedrov@saul.cis.upenn.edu

## Abstract

*We present a higher-order functional notation for polynomial-time computation with an arbitrary* $0,1$-*valued oracle. This formulation provides a linguistic characterization for classes such as* NP *and* BPP, *as well as a notation for probabilistic polynomial-time functions. The language is derived from Hofmann's adaptation of Bellantoni-Cook safe recursion, extended to oracle computation via work derived from that of Kapron and Cook. Like Hofmann's language, ours is an applied typed lambda calculus with complexity bounds enforced by a type system. The type system uses a modal operator to distinguish between two sorts of numerical expressions. Recursion can take place on only one of these sorts. The proof that the language captures precisely oracle polynomial time is model-theoretic, using adaptations of various techniques from category theory.*

## 1 Introduction

In 1964, Cobham proposed a characterization of feasible functions that is based on a binary-numeral form of primitive recursion [7]. In Cobham's definition, primitive recursion is restricted in an essentially ad hoc way, by requiring that any function defined by primitive recursion be bounded above by some other function already shown to be computable in polynomial time. Over the past 30+ years, Cobham's recursion scheme has been repeatedly analyzed and reworked. One motivation for this line of research has been to find a "logical" characterization of polynomial time that does not contain any obvious use of clocks or other mechanisms that count the number of computation steps. Another motivation has been to obtain a characterization of higher-order polynomial time [8].

We present a higher-order typed programming language characterizing polynomial-time computation with a $0,1$-valued oracle. Since probabilistic computation may be characterized as computation relative to an oracle, where the oracle is chosen probabilistically, we also view the language as characterizing probabilistic polynomial time computation. The principal complexity-theoretic property of the language is that every function (of a certain syntactic type) that is definable in the language can be computed in time that is bounded by a polynomial function of the input, independent of the oracle. The fact that the running time is bounded by the same polynomial, for all oracles, makes it possible to capture complexity classes such as NP and BPP. For example, NP is the class of languages $L$ for which there exists a particular kind of terms in the language. In particular, these terms must be such that for every candidate word $s$, there is some oracle $\phi$ whereby $t$, computing with the assistance of the oracle $\phi$, decides $s \in L$. Because we know that the language captures precisely the oracle polynomial time functions, we do not need to mention polynomial time explicitly in the definition.

For those not familiar with the area, it may be helpful to point out that there are several incomparable but equally compelling definitions of the class of computable functions of higher type (e.g., functions with function inputs). Therefore, we may also expect to find several apparently reasonable classes of higher-

*Partially supported by DoD MURI "Semantic Consistency in Information Exchange," ONR Grant N00014-97-1-0505, with additional support from NSF CCR-9629754.
†Additional support from Stanford University Fellowship.
‡Additional support from NSF Grant CCR-9800785.

order polynomial-time functions. (This general issue is discussed in [8], for example.) One natural approach to higher-order polynomial time is through programming languages that respect resource bounds. More specifically, suppose we can define a language that contains function symbols and such that every natural number function definable in the language can be computed in polynomial time. Then we may obtain a class of "higher-order polynomial time functions" by treating expressions in the language as functions of the higher-order variables they contain.

One complication that arises with time-bounded computation, but not with computability independent of resource bounds, is that computation time may depend on the values of the input function. For example, consider a function $f(x, g)$ with natural number input $x$ and function input $g$. Suppose that on input $x$ and $g$, the function $f$ applies some polynomial-time function $h$ to $g(x)$. If $g$ is some arbitrary input function, then we have no reason to expect the size of $g(x)$ to be bounded by some polynomial in $|x|$. If $g(x)$ is exponentially larger, for example, then the computation of $h(g(x))$ may be exponential, even if we just count the running time of $h$ and assume that $g(x)$ is obtained in a single step. For this reason, we restrict the oracle functions to taking only the value $0$ and $1$. (Of course, ordinary functions definable in the language may take arbitrary values.)

The starting point for the work presented here is a higher-order typed lambda calculus, containing function symbols of arbitrary type and a form of recursion operator called *safe recursion.* This calculus and associated complexity analysis were developed by Hofmann [11], building on work by Bellantoni and Cook [3, 5]. In brief, Bellantoni-Cook safe recursion achieves the same goal as Cobham's restricted form of primitive recursion, but through different means. Instead of an explicit bound, there is implicit control over complexity through the use of two separate lists of arguments. One list of arguments, called the *normal* arguments, may be used in any way. Arguments from the second list, referred to as *safe,* cannot be used as the recursion argument in any nested safe recursion. Through this mechanism, described in more detail in Section 3, it is possible to define all polynomial-time functions, but it is not possible to nest such computations a variable number of times. While Bellantoni and Cook worked in a first-order framework similar to ordinary primitive recursive notation, Hofmann captured the safe/normal distinction through a type system that brings the system closer to a convenient programming language. In particular, Hofmann's framework allows the use of higher-order functions.

Our extension of Hofmann's system retains the typed lambda calculus framework, but allows the use of an oracle function not assumed definable within the language. Since a nondeterministic or probabilistic machine uses a different "choice" or random bit at each branch point, we formulate our oracle primitive as a basic operation that returns the next bit of the oracle sequence each time it is called. This makes it easy to show how a probabilistic or nondeterministic algorithm can be written in our language.

Although it has little direct bearing on the results described here, our motivation for this work is the study of security properties. Specifically, as described in [14], we have developed a language for defining concurrent systems of probabilistic polynomial-time processes, with the sequential parts of each process written using the language described here. In this framework, the inherent complexity bounds allow us to quantify over all probabilistic polynomial-time adversaries by quantifying over processes expressible in the language. Related use of a language framework to quantify over adversaries has been developed in [1], but in a more abstract setting without complexity bounds.

## 2 Polynomial-time functionals

Nondeterministic and probabilistic Turing machines are usually defined as machines that may have more than one possible transition from a single configuration [19]. The difference between nondeterminism and randomness is not in the structure of the machine itself, but in the definition of acceptance: a nondeterministic machine accepts if there is *any* accepting computation, while probabilistic machines accept with probability determined by the number of coin flips along a computation path. It is easy to see that both forms of Turing machines are equivalent to deterministic Turing machines that use an oracle to decide which transition to take. Under the oracle-machine formulation, we would say that a "nondeterministic machine" accepts input $x$ if there exists some oracle (representing all nondeterministic choices) that allows it to accept $x$. Similarly, we may regard a probabilistic machine as an oracle Turing machine that consults a randomly chosen oracle. Because of the correspondence between branching computation and oracle computation, common complexity classes such as NP, PP and BPP are easily characterized using polynomial-time oracle computation. To be precise, we adopt the following definition:

**Definition 1.** A functional $f(\varphi, \vec{x})$, where $\varphi$ may be any function from $\mathbb{N}$ to $\{0, 1\}$, runs in *oracle polynomial time* if there exists a polynomial $p$ and an oracle Turing Machine $M$ whose output with oracle $\varphi$ and input $\vec{x}$ is $f(\varphi, \vec{x})$, and such that the running time of $M$ on inputs $\vec{x}$ is bounded by $p(|\vec{x}|)$, where $|\vec{x}|$ is the vector $|x_1|, \ldots, |x_n|$ and $|x_i| = \lceil \log_2 x_i \rceil$.

It is important to notice that the running time of $M$ must be bounded by a function of the length of the integer inputs, independent of the oracle. For this reason, the functions computable by oracle polynomial-time machines (as defined above) are different from the functions computable in polynomial time relative to any fixed oracle.

## 3 A language for oracle polynomial time

Our language OSLR is an extension of Hofmann's SLR with an oracle primitive. A central idea in SLR is that there are two types of natural number arguments to functions. Arguments of the first type, $\mathbf{N}$, are bounded numeric values whose length (number of bits) can only be an additive constant above any of the input values. Since arguments of type $\mathbf{N}$ are bounded, it is *safe* to pass them to nested recursive functions. Arguments of the second type, $\Box\mathbf{N}$, are *normal* natural number arguments that may be polynomially longer than input values of other functions. To avoid exponential-time computations, there are syntactic restrictions on primitive recursion that forbid use of normal arguments in the recursive position.

The types of SLR and OSLR are given by the grammar

$$\begin{array}{rcll} \tau & ::= & \mathbf{N} & \text{(natural numbers)} \\ & | & \tau \to \tau & \text{(function from safe input)} \\ & | & \Box\tau \to \tau & \text{(functions from unsafe input)} \end{array}$$

It should be noted here that $\Box\,\mathbf{N}$ is not actually an SLR type. There are two explanations for this situation, both equally valid. The simpler explanation is that Hofmann has modeled in a type system precisely what Bellantoni presented in a different framework. In particular, as explained in Section 5.1, Bellantoni makes use of input parameters of two sorts. There is only one sort of output in his framework, although there are syntactic restrictions on the places in which the output can be used. Hence, in SLR there is never a modality on the output type of a function. (The natural numbers are a special case; one may think of them as functions of no arguments.) The typing rules enforce the syntactic restrictions on composition.

Alternatively, we may employ an explanation derived from the theory of modal logics. In particular, the distinction between $\Box\tau$ and $\tau$ is related to modal operators. Originally inspired by type systems derived from linear logic [10], similar type distinctions have been used in program analysis and compilation to characterize the time at which a value becomes known [9, 20]. From the perspective of modal logic, there *do* exist modal output types, and consequently, there is a $\Box\,\mathbf{N}$ type.

One contribution of [11] is the limited way that the modality $\Box$ may occur in types. This avoids the expression forms associated with ! in linear logic and, more generally associated with any modal type operator associated with any monad [18]. A second innovation we adopt from [11] is a form of subtyping, with $A \to B <: \Box A \to B$, further avoiding explicit conversions between types. Together, these innovations allow a useful form of type inference [11]: there is a type-checking algorithm that can automatically determine the type of any expression, without requiring the distinction between $\mathbf{N}$ and $\Box\mathbf{N}$ to be written into expressions. (Since OSLR uses the same overall type system as SLR, this algorithm carries over to OSLR.) Thus, from the point of view of modal logic, the type $\Box\,\mathbf{N}$ exists, but the type-checking algorithm removes any need for its use.

The expressions of OSLR are given by the following grammar, where $v$ may be any variable and $\tau$ any type:

$$\begin{array}{rcll} e & ::= & v & \text{(variable)} \\ & | & n & \text{(numeral)} \\ & | & \mathbf{S_0} \mid \mathbf{S_1} & \text{(doubling functions)} \\ & | & (e_1\ e_2) & \text{(application)} \\ & | & \mathbf{fun}(v : \tau)\ e & \text{(abstraction)} \\ & | & \mathbf{case}_\tau\ e_1\ \mathbf{zero}\ e_2 & \\ & & \mathbf{even}\ e_3\ \mathbf{odd}\ e_4 & \text{(case distinction)} \\ & | & \mathbf{saferec} & \text{(safe recursion))} \\ & | & \mathbf{rand} & \text{(oracle bit)} \end{array}$$

Variables, lambda abstraction and application are standard from typed lambda calculus (see, e.g., [17]), with the modification that $\mathbf{fun}(v : \tau)\ e$ may have types $\tau \to \sigma$ or $\Box\tau \to \sigma$, according to the type inference algorithm [11]. In particular, a function gets the former type if and only if the argument of type $\tau$ is not passed to any function expecting a normal input. Functions $\mathbf{S_0}$ and $\mathbf{S_1}$ double a number or double and add 1, and $\mathbf{case}_\tau$ has three branches, according to whether the first argument is zero, odd, or even. The restricted primitive recursion operator $\mathbf{saferec}$ is described below. The function $\mathbf{rand}$ returns the next bit from the oracle, with repeated calls potentially returning different bits. (There is nothing about this language that requires the oracle to be chosen randomly, but we use $\mathbf{rand}$ for oracle access since our primary interest is in probabilistic polynomial time.)

The type system is an extension of standard typed lambda calculus, with subtyping as described above and restrictions on computation achieved by careful distinction between $\mathbf{N}$ and $\Box\mathbf{N}$ in the typing of basic operations. The types of constants are as follows:

$$\begin{array}{rcl} n & : & \mathbf{N}, \text{ when } n \text{ is an integer constant} \\ \mathbf{S_0} & : & \mathbf{N} \to \mathbf{N} \\ \mathbf{S_1} & : & \mathbf{N} \to \mathbf{N} \\ \mathbf{case}_\tau & : & \mathbf{N} \to \tau \to (\mathbf{N} \to \tau) \to (\mathbf{N} \to \tau) \to \tau \\ \mathbf{saferec} & : & \Box\mathbf{N} \to \mathbf{N} \to (\Box\mathbf{N} \to \mathbf{N} \to \mathbf{N}) \to \mathbf{N} \\ \mathbf{rand} & : & \mathbf{N} \end{array}$$

Intuitively, we would expect $n : \Box\mathbf{N}$ for numeral $n$, since an explicit numeral has a fixed value, and therefore cannot implicitly define a fast-growing function of any input. However, $\Box\mathbf{N}$ itself is not a type. Instead, the typing rules of [11] are formulated so that it is possible to apply a function of type $\Box\mathbf{N} \to \mathbf{N}$ to a numeral, since a numeral does not have any non-modal free variables.

The intending meaning of the **case** construct is that

$$\mathbf{case}\ n\ \mathbf{zero}\ e_1\ \mathbf{even}\ e_2\ \mathbf{odd}\ e_3 = \begin{cases} e_1 & \text{if } n = 0 \\ e_2 n & \text{if } n \text{ is even} \\ e_3 n & \text{if } n \text{ is odd} \end{cases}$$

The intended meaning of **saferec** is that

$$\mathbf{saferec}\ n\ a\ f = \begin{cases} a & \text{if } n = 0 \\ f\ n\ (\mathbf{saferec}\ \lfloor n/2 \rfloor\ a\ f) & \text{otherwise} \end{cases}$$

The type of $\mathbf{saferec}$ captures the $B^*$ requirements on predicative recursion described in Section 5.1. In particular, the output of a sub-recursion is presented to $f$ in a safe position, i.e., as a $\mathbf{N}$ rather than a $\Box\,\mathbf{N}$ argument.

Note that **rand** has side-effects. (In particular, successive uses of **rand** return successive oracle bits, so we may view a use of **rand** as incrementing a counter that indicates which oracle bit should be used as the value of the next use of **rand**.) The presence of side-effects means that the order in which we perform reductions on terms can influence the value of those terms. For example, consider:

$$(\mathbf{fun}(x : N)(\texttt{plus}\ x\ x))\ \mathbf{rand}$$

where $\texttt{plus} : \Box\,\mathbf{N} \to \Box\,\mathbf{N} \to \mathbf{N}$ is the usual addition function. Under a “lazy” evaluation scheme, this term would evaluate to 1, if one of the next two oracle bits was 0 and the other was 1. We choose a LISP-like call-by-value evaluation scheme, feeling that it is the most “natural.” Under our semantics, the term above can evaluate only to 0 or 2, but never to 1, because the evaluation of $\mathbf{rand}$ happens *before* the function application. A complete set-theoretic semantics for `OSLR` is given below.

It is worth mentioning one alternate language design that we considered. Instead of accessing an oracle bit-by-bit using $\mathbf{rand} : \mathbf{N}$, we could allow “random access” to the entire oracle by including a function $\mathbf{oracle} : \mathbf{N} \to \mathbf{N}$ instead. At first glance, it might seem that the second is more general. However, it is easy to write a small loop that reads some polynomial number of oracle bits using $\mathbf{rand} : \mathbf{N}$ and concatenates them into an integer value for later use. In contrast, we were not able to find any direct way translation in the opposite direction. Specifically, many randomized algorithms can be written fairly directly in `OSLR` using a “next random bit” primitive $\mathbf{rand} : \mathbf{N}$. When we attempted to find syntactic transformations that produced an equivalent algorithm using an oracle function $\mathbf{oracle} : \mathbf{N} \to \mathbf{N}$, we found that some artifacts of the type of $\mathbf{saferec}$ made it difficult to maintain a bit counter (indicating the next oracle bit to access) and pass this into and out of primitive recursive functions. We therefore decided to make a “next random bit” primitive $\mathbf{rand} : \mathbf{N}$ a basic function of `OSLR` and prove that every function definable using $\mathbf{rand}$ is computable in polynomial time.

## 4 A Set-Theoretic Semantics for `OSLR`

We begin by defining a set-theoretic interpretation $\mathbf{Set}[\![\cdot]\!]$ on types. First, we define a mapping $\overline{\mathbf{Set}}[\![\cdot]\!]$ as follows:

$$\begin{array}{rcl} \overline{\mathbf{Set}}[\![\mathbf{N}]\!] & = & \mathbb{N} \\ \overline{\mathbf{Set}}[\![A \to B]\!] & = & \overline{\mathbf{Set}}[\![A]\!] \times \mathbb{N} \to \overline{\mathbf{Set}}[\![B]\!] \times \mathbb{N} \\ \overline{\mathbf{Set}}[\![\Box A \to B]\!] & = & \overline{\mathbf{Set}}[\![A]\!] \times \mathbb{N} \to \overline{\mathbf{Set}}[\![B]\!] \times \mathbb{N} \end{array}$$

Given $\overline{\mathbf{Set}}[\![\cdot]\!]$ we can define $\mathbf{Set}[\![\ ]\!]$

$$\begin{array}{rcl} \mathbf{Set}[\![\mathbf{N}]\!] & = & \mathbb{N} \to \mathbb{N} \times \mathbb{N} \\ \mathbf{Set}[\![A \to B]\!] & = & \overline{\mathbf{Set}}[\![A \to B]\!] \\ \mathbf{Set}[\![\Box A \to B]\!] & = & \overline{\mathbf{Set}}[\![\Box A \to B]\!] \end{array}$$

An *environment* $\eta$ is a mapping from variables in $V$ to $\cup_{t \in T}\mathbf{Set}[\![t]\!]$ together with a single (possibly non-recursive) function $\varphi : \mathbb{N} \to \{0, 1\}$.

Next, we define a set-theoretic interpretation $\mathbf{Set}[\![\cdot]\!]\eta$ of a typing judgment $\Gamma \rhd e : A$ in an environment $\eta$. (This interpretation will be such that that $\mathbf{Set}[\![\Gamma \rhd e : A]\!]\eta$ will be an element of $\mathbf{Set}[\![A]\!]$.) The definition is by induction is on typing judgments. First, we handle the T-CONST rule:

$$\begin{array}{rcl} \mathbf{Set}[\![\Gamma \rhd \mathtt{n} : \mathbf{N}]\!]\eta & = & \lambda x.\langle n, x\rangle \\ \mathbf{Set}[\![\Gamma \rhd \mathbf{S_0} : \mathbf{N} \to \mathbf{N}]\!]\eta & = & \lambda\langle x, y\rangle.\langle 2x, y\rangle \\ \mathbf{Set}[\![\Gamma \rhd \mathbf{S_1} : \mathbf{N} \to \mathbf{N}]\!]\eta & = & \lambda\langle x, y\rangle.\langle 2x+1, y\rangle \\ \mathbf{Set}[\![\Gamma \rhd \mathbf{rand} : \mathbf{N}]\!]\eta & = & \lambda x.\langle \phi(x), x+1\rangle \end{array}$$

Here we write $n$ for the number corresponding to the numeral $\mathtt{n}$. The definition for **saferec** is a bit more complex. It is the functional $S$ given by:

$$S = \lambda\langle n, a\rangle.\langle\lambda\langle x, b\rangle.\langle\lambda\langle f, c\rangle.g, b\rangle, a\rangle$$

where $g$ is:

$$\begin{cases} \langle x, c\rangle & (n = 0) \\ \text{let } d = \pi_1(f\langle n, c\rangle) \text{ in} & \\ \quad (\pi_0(f\langle n, c\rangle))(S\langle\lfloor n/2\rfloor, d\rangle\langle x, d\rangle\langle f, d\rangle) & (n > 0) \end{cases}$$

The rules for variables and subtyping are simple:

$$\begin{array}{rcl} \mathbf{Set}[\![\Gamma \rhd x : \Gamma(x)]\!]\eta & = & \lambda n.\langle\eta(x), n\rangle \\ \mathbf{Set}[\![\Gamma \rhd e : B]\!]\eta & = & \mathbf{Set}[\![\Gamma \rhd e : A]\!]\eta \end{array}$$

For abstraction, we take

$$\mathbf{Set}[\![\Gamma \rhd \mathbf{fun}(x : A)e : A \to B]\!]\eta$$

to be the unique function $f : \mathbf{Set}[\![A]\!] \times \mathbb{N} \to \mathbf{Set}[\![B]\!] \times \mathbb{N}$ such that $f(v, n) = \mathbf{Set}[\![\Gamma, x : A \rhd e : B]\!]\,\{\eta, x \mapsto v\}\,(n)$. The interpretation

$$\mathbf{Set}[\![\Gamma \rhd \mathbf{fun}(x : A)e : \Box A \to B]\!]\eta$$

is precisely analogous.

For application, the situation is somewhat more complex. If the function is of type $\mathbf{N} \to \mathbf{N}$, we have:

$$\begin{array}{l} \mathbf{Set}[\![\Gamma, \Delta_1, \Delta_2 \rhd (e_1 e_2) : \mathbf{N}]\!]\eta = \\ \quad \lambda n.(\mathbf{Set}[\![\Gamma, \Delta_1 \rhd e_1 : A \overset{a}{\to} B]\!]\eta(\langle v, n'\rangle)) \end{array}$$

where

$$\langle v, n'\rangle = (\mathbf{Set}[\![\Gamma, \Delta_2 \rhd e_2 : A]\!]\eta(n))$$

The situation for more complex types is completely analogous. In each case the argument is evaluated before the application is performed.

We have omitted the set-theoretic interpretation for case-distinction here, but it is straight-forward in light of the above.

The set-theoretic semantics for OSLR is non-trivial in the sense that, for instance, numerals are not interpreted as elements of $\mathbb{N}$. The addition of the store (the counter keeping track of what oracle bit is next) is repeated in a categorical setting below.

## 5 An equivalence in five easy pieces

We prove an exact correspondence between OSLR functions of type $\Box\,\mathbf{N} \to \mathbf{N}$ and the oracle polynomial-time functionals.

**Theorem 1.** *The oracle polynomial time functionals are precisely the* OSLR *terms of type* $\Box\,\mathbf{N}^m \to \mathbf{N}^n \to \mathbf{N}$.

The proof requires five steps, four extending previous results to oracle computation and one (step 4) involving the oracle mechanism specifically:

1. *Define class $B^*$:* The syntactic class $B^*$ is an extension of Bellantoni-Cook's class $B$, corresponding to oracle polynomial-time functions.

2. *Construct base category $\mathcal{C}$ of polynomial-time functionals.* Following a standard categorical construction also used in [11], we form a category $\mathcal{C}$ from oracle polynomial time functions. The purpose is to apply model-theoretic techniques from category theory and avoid laborious operational reasoning about the evaluation of higher-order OSLR expressions.

3. *Embed $\mathcal{C}$ in category $\hat{\mathcal{C}}$.* Following another standard categorical construction, we embed our category of oracle polynomial-time functions in a

larger category $\hat{\mathcal{C}}$ that allows us to interpret the higher-order types of OSLR. The standard construction is that $\hat{\mathcal{C}}$ is the category of presheaves over $\mathcal{C}$. Since we have two sorts of basic natural number expressions, the standard Yoneda lemma [2] cannot be applied. However, a variant of it (also used by Hofmann) can be used to show a form of conservativity of $\hat{\mathcal{C}}$ over $\mathcal{C}$.

4. *Form a Kleisli category $\mathcal{K}$ over category* $\hat{\mathcal{C}}$. The Kleisli construction [15, 18] is a standard technique for extending a semantic framework with additional structure or "side information" that is passed automatically from one function to another when they are composed. By choosing an appropriate monad, we use this construction to give an semantic interpretation to the implicit counter used to maintain an index into the infinite bit string provided by the oracle.

5. *Prove bijective correspondence.* The final step is to show that for every map in the Kleisli category (model of OSLR), there is an (equivalent) oracle polynomial-time functional, and conversely. This uses a categorical version of the traditional logical relations argument from typed lambda calculus [17].

It is worth emphasizing that this is a model-theoretic proof, using classes of functions rather than algorithms. In particular, the theorem states only that each definable functional is computable in polynomial time. The proof does not provide an algorithm for evaluating expressions within this time bound in a step-by-step fashion, although we believe an appropriate machine model could be derived using [3, 5].

The use of category theory, in [11] and here, is motivated by the fact that models involved form what are known as "non-well-pointed" categories. Intuitively, this means that there is some intensional structure involved that is not easily captured in the classical model theory based on Henkin models. While it is difficult to explain the mathematical reasons in elementary terms, one specific issue involves the bijection between functions in the model and oracle polynomial-time functionals in Step 5. The non-categorical alternative would be to use logical relations over Henkin models [17]. However, these are uniquely determined by their extension at ground type, while the proof in Step 5 requires a careful choice of relation at function types. The presheaf and Kleisli category constructions are also standard categorical techniques that do not seem to have natural non-categorical analogs based on Henkin models.

## 5.1 Safe recursion with oracle

In [3], Bellantoni defines a class $B$ of functions and shows that $B$ characterizes precisely the polynomial time functions. Addition of $0, 1$-valued oracles is considered in [4]. This leads to a class $B^*$ of functionals that characterize oracle polynomial time. The much harder problem of allowing arbitrary oracle functions is considered in [13], but has not been reformulated using safe recursion.

As mentioned earlier, class $B$ (and $B^*$) contain functions with two forms of integer arguments, called *safe* and *normal.* Normal arguments will be written to the left of safe arguments, separated by a semicolon. For example, $f(x, y; z)$ indicates a function of two normal arguments and one safe argument. The functions in $B$ (and $B^*$) may perform any polynomial time operation on their normal arguments, but may only perform operations on safe inputs that do not increase the length of the output by more than additive constant.

The functionals in $B^*$ accept one $0, 1$-valued function (oracle) as input, in addition to safe and normal integer arguments. The oracle argument is written first, so that $f(\varphi; x, y; z)$ is a functional whose arguments are the oracle $\varphi$, the normal arguments $x$ and $y$ and the safe argument $z$. The class $B^*$ is the smallest class containing:

- The constant $0 + 0$-ary function $0(\varphi; ;) = 0$,
- For each $m$ and $n$, the $m+n$-ary projection functions
$$\pi_j^{m,n}(\varphi; x_1, \ldots, x_m; x_{m+1}, \ldots, x_{m+n}) = x_j,$$
- The successor functions $s_i(\varphi; ; y) = 2y + i$ for $i \in \{0, 1\}$,
- The predecessor function $p(\varphi; ; y) = \lfloor y/2 \rfloor$,
- Oracle application to safe argument $\mathrm{Ap}(\varphi; ; x) = \varphi(x)$.

and closed under the following schemas:

- The "predicative recursion" schema, allowing the definition of a new function $f$ as

$$f(\varphi; a, \vec{x}; \vec{y}) = \begin{cases} g(\varphi; \vec{x}; \vec{y}) & (a = 0) \\ h_i(\varphi; \lfloor a/2 \rfloor, \vec{x}; \vec{y}, f(\varphi; \lfloor a/2 \rfloor, \vec{x}; \vec{y})) & (a > 0) \end{cases}$$

  if $g, h_i$ are in $B^*$ for $i \in \{0, 1\}$,

- The "safe composition" schema, allowing the definition of a new function $f$ as

$$f(\varphi; \vec{x}; \vec{y}) = h(\varphi; \vec{r}(\varphi; \vec{x};); \vec{t}(\varphi; \vec{x}; \vec{y})),$$

  when $h$, $\vec{r}$, and $\vec{t}$ are in $B^*$.

**Proposition 1 (Bellantoni).** *The functionals definable in $B^*$ correspond precisely to the oracle polynomial time functionals. In particular, every functional in $B^*$ may be computed in oracle polynomial time, and all oracle polynomial time functionals may be expressed in $B^*$.*

### 5.2 The category $\mathcal{C}$

We begin by forming a category of oracle polynomial-time functions under safe composition. The objects of category $\mathcal{C}$ are pairs of natural numbers, indicating an arity (number of normal inputs and number of safe inputs). The maps, or morphisms, of the category are tuples of oracle polynomial-time functions containing the correct number of functions to allow us to compose correctly. More specifically, a morphism from $(m, n)$ to $(m', n')$ is a pair $< f_1^1, \ldots, f_{m'}^1; f_1^2, \ldots, f_{n'}^2 >$ of sequences of functions, with each $f_i^1$ an $m$-ary functional in $B^*$ and each $f_j^2$ an $m+n$-ary functional in $B^*$. The identity morphism at $(m, n)$ is the obvious tuple of $m+n$-ary projections. Composition proceeds according to safe composition in $B^*$.

### 5.3 The category $\hat{\mathcal{C}}$

A standard method for embedding a set of first-order functions into a model of a higher-order typed language, without introducing additional first-order functions, is the Yoneda embedding [2]. This involves a correspondence between a category $\mathcal{D}$ and the category $\hat{\mathcal{D}}$ of presheaves (contravariant functors into $\mathbf{Set}$) over $\mathcal{D}$. Since our aim is to prove properties of a higher-order language built from polynomial-time functions, this tool is an obvious one to use, at least for category theorists. However, we must adapt the standard proof to account for the fact that our initial functions have oracle, normal and safe arguments. Specifically, we let $\hat{\mathcal{C}}$ be the presheaf category over $\mathcal{C}$.

An elementary fact of category theory is that presheaf categories are cartesian closed. This allows us to interpret every SLR/OSLR type as an object of this category, namely:

$$\begin{aligned} [\![\mathbf{N}]\!] &= \mathrm{Hom}_{\mathcal{C}}(-, (0, 1)) \\ [\![A \to B]\!] &= [\![A]\!] \Rightarrow [\![B]\!] \\ [\![\Box A \to B]\!] &= \Box[\![A]\!] \Rightarrow [\![B]\!] \end{aligned}$$

Here, $\Box F$, for any functor $F$, is the functor that takes $(m, n)$ to $F(m, 0)$.

We follow standard techniques to give denotations for the expressions of pure SLR. In particular, for every typing judgment $\Gamma \vdash e : \tau$, indicating that $e$ is a well-formed term over a set of variables listed in $\Gamma$, there is a corresponding morphism $[\![\Gamma \vdash e : \tau]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$ in $\hat{\mathcal{C}}$. (Here, $[\![\Gamma]\!]$ is understood to be the product of the types of the variables enumerated in $\Gamma$.)

We are able to reuse a proof of Hofmann to obtain the following correspondence, analogous to the standard Yoneda Lemma [2]:

**Proposition 2 (Hofmann).** *There exists a bijection between the set of natural transformations from $[\![\Box\, \mathbf{N}]\!]^m \times [\![\mathbf{N}]\!]^m$ to $[\![\mathbf{N}]\!]$ and the set of $m+n$-ary functionals in $B^*$.*

### 5.4 The category $\mathcal{K}$

While $\hat{\mathcal{C}}$ forms a semantic model of SLR, there is no immediate way to access the oracle. More specifically, need some sort of mechanism for maintaining an implicit counter and accessing the next bit of the oracle for each occurrence of $\mathbf{rand}$. The technique we apply comes from a general approach to imperative languages in a categorical framework. Specifically, we use a monad-based technique first identified in [18]. Intuitively, a Kleisli category over a category $\mathcal{D}$ endows each map (morphism) of $\mathcal{D}$ with some extra structure and redefines function composition so that this extra structure is preserved. For the purpose of carrying out this construction, we define a monad $\mathcal{M} = (T, \eta, \mu)$ over $\hat{\mathcal{C}}$. The functor $T : \hat{\mathcal{C}} \to \hat{\mathcal{C}}$ adding structure to each type is $[[\![\mathbf{N}]\!] \Rightarrow C \times [\![\mathbf{N}]\!]]$, since the additional structure we require is a natural number index into the oracle sequence.

The Kleisli category $\mathcal{K}$ over $\mathcal{M}$ has, as objects, the objects of $\mathcal{C}$. However, a morphism from $A$ to $B$ in $\mathcal{K}$ is a $\hat{\mathcal{C}}$ morphism from $A$ to $TB$, i.e., a morphism from $A$ to $[\llbracket \mathbf{N} \rrbracket \Rightarrow B \times \llbracket \mathbf{N} \rrbracket]$. The natural transformation $\eta : 1_{\hat{\mathcal{C}}} \stackrel{\circ}{\to} T$ provides a way of "lifting" values from $\hat{\mathcal{C}}$ into $\mathcal{K}$. Intuitively, $\eta$ gives a way of converting values of type $A$ to objects of type $T(A)$. The natural transformation $\mu : T^2 \stackrel{\circ}{\to} T$ provides a means of doing composition. In particular, if $f : A \to B$ and $g : B \to C$ are two arrows in $\mathcal{K}$, then $f; g : A \to C$ is given by $f; Tg; \mu_C$. (This is, of course, an arrow from $A$ to $TC$ in $\hat{\mathcal{C}}$, as desired.) The particular composition used performs the obvious threading of stores; the store output by $f$ is given as input to $g$, and it is the resulting store output by $g$ that is output by the composition $f; g$.

### 5.5 Conservativity

Finally, we must demonstrate the existence of a bijection between `OSLR` terms of type $\Box\,\mathbf{N}^m \to \mathbf{N}^n \to \mathbf{N}$ and elements of $B^*$ and show that each expression denotes the appropriate oracle polynomial-time function. The full proof requires construction of a categorical logical relation by the sconing or Freyd cover technique and is too lengthy to repeat here. However, we can give a brief sketch of the underlying bijection.

First, assume that we have an appropriately typed `OSLR` expression $f$. The denotation ${}^{\mathcal{K}}\llbracket \vdash\ f\ :\ \Box\,\mathbf{N}^m \to \mathbf{N}^n \to \mathbf{N} \rrbracket$ in category $\mathcal{K}$ is an arrow from $1_{\hat{\mathcal{C}}}$ to $T(\llbracket \Box\,\mathbf{N}^m \to \mathbf{N}^n \to \mathbf{N} \rrbracket)$, i.e, $[\mathbf{N} \Rightarrow \llbracket \Box\,\mathbf{N}^m \to \mathbf{N}^n \to \mathbf{N} \rrbracket \times \mathbf{N}]$. This type indicates that the semantic type of $f$ is a function that takes an initial oracle-counter value, the normal and safe numeric arguments, and then returns the function value and a counter value indicating the number of oracle bits used. The counter values are needed if we wish to compose this function with others, but if this is the entire function we wish to compute, then we can begin with counter value 0 and discard the output counter value. A basic property of function objects in cartesian closed categories allows us to "evaluate" this function at the point $\llbracket \vdash\ 0 : \mathbf{N} \rrbracket$ to obtain an arrow from $1_{\mathcal{C}}$ to $\llbracket \Box\,\mathbf{N}^m \to \mathbf{N}^n \to \mathbf{N} \rrbracket \times \llbracket \mathbf{N} \rrbracket$, thereby setting the initial counter value to 0. We may project out the function value and discard the final counter value obtaining, by Lemma 2 above, a function of type $\Box\,\mathbf{N}^m \to \mathbf{N}^n \to \mathbf{N}$, i.e., an element of $B^*$.

The other direction is simpler. Most of the $B^*$ constants and schemas have direct `OSLR` analogs. However, some argument must be made to show that `OSLR` can simulate $B^*$'s oracle access $\mathrm{Ap}$, using the `OSLR` **rand** primitive. By Theorem 1, each functional of $B^*$ has a polynomial bound, depending only on the length of the integer inputs. Therefore, as outlined at the end of Section 3, the `OSLR` function may request a polynomial number of oracle bits initially and then access any needed bit by simple bounded numeric operations.

## 6 Conclusion and Related Work

In this paper, we have presented a higher-order typed lambda-calculus `OSLR` with the property that the definable functions of type $\Box\mathbf{N} \to \mathbf{N}$ are exactly the oracle polynomial-time functionals. In contrast to the Bellantoni-Cook notation for safe recursion, the language allows definition and use of higher-order functions of any type. In this respect, our language, more closely resembles a complexity-sensitive version of the typed programming languages ML [16] than a mathematical notation for recursive functions.

As in Hofmann's `SLR` [11], the complexity restrictions of `OSLR` are enforced by a type system. The type system is entirely standard, except for the unary operator $\Box$ which is used to distinguish between natural numbers that can be used as recursion arguments and natural numbers that cannot. The distinction between type $\Box\tau$ and type $\tau$ has a precise correspondence with the distinction between modal and non-modal formulas in modal logic. This correspondence is an instance of the well-known *Curry-Howard Isomorphism* between constructive logics and typed functional languages [12, 17]. Like modal operators in type systems used for binding-time analysis and other forms of program analysis (e.g., [9, 20]), the typing rules for $\Box\tau$ in `SLR` and `OSLR` have the properties of the necessitation rule from modal logic. In this sense, Hofmann's characterization of polynomial time and our characterization of oracle polynomial time are logical characterizations of these complexity classes.

Bellantoni, Niggl and Schwichtenberg are also studying linguistic characterizations of higher-order polynomial time [6]. One difference between their work and ours is the language itself: ours is a standard lambda calculus with one twist in the type system, while their proposals have thus far involved various syntactic restrictions.

*Acknowledgments:* Thanks to Stephen Bellantoni, Dan Boneh, Viviana Bono, Peter Freyd, Stefano Guerrini, Martin Hofmann, Sampath Kannan, Bruce Kapron, Dusko Pavlovic, and Jim Royer for helpful discussions and advice on relevant literature.

## References

[1] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. In *Proc. 4th ACM Conference on Computer and Communications Security*, pages 36–47, 1997. Revised and expanded versions to appear in *Information and Computation* and as SRC Research Report 149 (January 1998).

[2] M. Barr and C. Wells. *Category theory for computing science*. Prentice Hall International, 1990.

[3] S. Bellantoni. *Predicative Recursion and Computational Complexity*. PhD thesis, University of Toronto, 1992.

[4] S. Bellantoni. Predicative recursion and the polytime hierarchy. In P. Clote and J. Remmel, editors, *Feasible Mathematics II*, pages 15–29. Birkhauser, 1995.

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

[6] S. Bellantoni, K.-H. Niggl, and H. Schwichtenberg. Untitled manuscript, March 1998.

[7] A. Cobham. The intrinsic computational difficulty of functions. In *Proc. Int'l Cong. Logic Methodology and Philosophy of Science*, pages 24–30. North-Holland, 1964.

[8] S. Cook. Computability and complexity of higher-type functions. In Y. Moschovakis, editor, *Logic from Computer Science*, pages 51–72. Springer-Verlag, 1992.

[9] R. Davies and F. Pfenning. A modal analysis of staged computation. In *23rd Annual ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 258–270, 1996.

[10] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[11] M. Hofmann. A mixed/modal lambda calculus with applications to Bellantoni-Cook safe recursion. In M. Nielsen and W. Thomas, editors, *Computer Science Logic, 11-th International Workshop, CSL '97, Annual Conference of the EACSL, Aarhus, Denmark, August 23-29, 1997, Selected Papers*. Springer LNCS Vol. 1414, 1998.

[12] 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, 1980.

[13] B. Kapron and S. Cook. A new characterization of type-2 feasibility. *SIAM J. Computing*, 25(1):117–132, 1996.

[14] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *ACM Conference on Computer Security*, 1998.

[15] S. MacLane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin, 1971.

[16] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[17] J. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[18] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. Preliminary version appeared in *Proc. IEEE Symp. on Logic in Computer Science,* 1989, under the title Computational lambda calculus and monads.

[19] M. Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[20] P. Wickline, P. Lee, F. Pfenning, and R. Davies. Modal types as staging specifications for run-time code generation. *ACM Surveys: Special Issue on Partial Evaluation*, page (To appear), 1998.